Lecture notes

Numerical Mathematics 1

Clemens Kirisits

Lukas Exl

September 30, 2024

Contents

Chapter 1

Basic Concepts of Numerical Analysis

Numerical analysis is the study of algorithms for solving problems of mathematical analysis. Its importance and usefulness become particularly apparent when regarding it as a subdiscipline of applied mathematics. Frequently, solving a problem in applied mathematics includes (some of) the following steps.

Modelling. A real-world problem is translated into a mathematical one. For example, the distribution of heat in a certain material can be modelled by a partial differential equation called heat equation.

Analysis. The mathematical problem is analysed. This step often involves the question of well-posedness: Does the problem have a unique solution that depends continuously on the data?

Approximation. Very often, solutions to mathematical problems cannot be computed directly. The problem must be approximated by a simpler one. Discretization, i.e. replacing continuous objects by discrete counterparts, is an important example of an approximation process.

Algorithm. The reduced problem is broken down into an algorithm: a sequence of simple steps that can be followed through mechanically.

Software. Software must be written so that the algorithm can be executed on a computer.

Validation. If it is possible to make measurements of the real-world phenomenon, then these measurements can be used to validate the mathematical model by comparing them to the computer output.

Prediction. Finally, the model can be used to gain new insights about the original problem that would be too costly or even impossible to obtain otherwise.

Numerical analysis mainly occupies the steps *Approximation* and *Algorithm*,[1] thereby bridging the gap between mathematical problems and computers. That

[1] However, writing software is an important aspect of numerical analysis as well, since it allows you, for instance, to compare the theoretical properties of an algorithm with its practical behaviour.

is, it addresses the questions (i) how mathematical problems can be turned into a form solvable by a computer, and (ii) how numerical solutions can be computed efficiently and accurately.

Another way of looking at numerical analysis is in terms of errors. Suppose, while validating your model for a particular problem, you realize that your computer output is significantly wrong. In this case the observed errors might have the following sources.

Data uncertainty. Many algorithms are fed on data produced by real-world measurements or by previous computations. In both cases the data can only be accurate to a certain degree.

Modelling. No mathematical model is a perfect description of reality. Therefore modelling inaccuracies must be taken into account.

Discretization. Since discretization, or truncation, is a kind of approximation, it typically introduces errors. These errors are called truncation errors.

Roundoff. On computers real numbers are replaced by floating point numbers. Therefore, both data input and internal computations lead to roundoff errors.

The focus of numerical analysis usually lies on controlling truncation and roundoff errors.

This chapter's narrative is as follows. Computers approximate real numbers with a finite set of rational numbers, so-called floating point numbers (Sec. 1.2). Therefore, computer calculations are inexact in general. A notoriously problematic operation is subtraction of two almost equal numbers (Sec. 1.3). More generally, the concept of conditioning (Secs. 1.4, 1.5) helps us assess the sensitivity of a problem to small perturbations such as roundoff errors. On the other hand, since floating point arithmetic is inexact, mathematically equivalent algorithms lead to different results. Therefore we have to be able to tell which algorithms are stable, i.e. robust with respect to roundoff errors, and which are not (Sec. 1.6).

1.1 Absolute and Relative Errors

In the following we often want to assess the quality of an approximation $\tilde{x}$ to a number x. Two very common measures are the *absolute error* of $\tilde{x}$

$$|\Delta x| := |x - \tilde{x}|$$

and, for $x \neq 0$, the *relative error* of $\tilde{x}$

$$|\delta x| := \frac{|x - \tilde{x}|}{|x|} = \frac{|\Delta x|}{|x|}.$$

Note that, if there is a number ϵ such that $\tilde{x} = x(1+\epsilon)$, then $|\epsilon| = |\delta x|$. In many situations the relative error is more informative than the absolute one. One reason is its *scale-invariance*: Replacing x and $\tilde{x}$ by αx and $\alpha\tilde{x}$, for some $\alpha > 0$, yields the same relative error.

For vectors and matrices the absolute and relative errors are defined using norms

$$\|\Delta x\| := \|x - \tilde{x}\|, \quad \|\delta x\| := \frac{\|\Delta x\|}{\|x\|},$$
$$\|\Delta A\| := \|A - \tilde{A}\|, \quad \|\delta A\| := \frac{\|\Delta A\|}{\|A\|}.$$

Another possibility is to consider *componentwise errors*, for instance,

$$|\delta x_i| = \frac{|x_i - \tilde{x}_i|}{|x_i|},$$
$$|\delta a_{ij}| = \frac{|a_{ij} - \tilde{a}_{ij}|}{|a_{ij}|}.$$

1.2 Floating Point Arithmetic

Designing and analysing good algorithms requires knowledge of how computers handle numbers. The following example is taken from [6].

Example 1.1 (3-digit calculator)**.** Suppose you have a very simple calculator which represents numbers in the following way:

$$x = \pm 0.d_1d_2d_3 \times 10^e, \quad \text{where} \begin{cases} d_1 & \in \{1,\dots,9\} \\ d_2 & \in \{0,\dots,9\} \\ d_3 & \in \{0,\dots,9\} \\ e & \in \{-9,\dots,9\}. \end{cases}$$

A few observations are in order:

- The first digit d_1 does not take the value 0 for reasons of uniqueness. If we allowed $d_1 = 0$, then we could write certain numbers in more than one way, e.g. $74 = 0.74 \times 10^2 = 0.074 \times 10^3$.
- Clearly, zero must be representable. However, since $d_1 \neq 0$, we need an exception to do so, for example $d_1 = d_2 = d_3 = e = 0$.
- Our calculator can only represent finitely many numbers: $2 \times 9 \times 10 \times 10 \times 19 + 1 = 34201$.
- In particular, there is a largest number (0.999×10^9) and a smallest positive number (0.100×10^{-9}). Notice that with the convention $d_1 \neq 0$ we have "lost" the even smaller numbers of the form $0.0d_2d_3 \times 10^{-9}$.

- The calculator's precision depends on the number of digits, which in our case is three. In order to represent the number 123456, for example, the toy calculator cannot do better than 0.123×10^6. The resulting relative error is of order 10^{-3}.
- Similarly, results of calculations generally will have to be rounded in order to fit the 3-digit format. For instance, $(0.123 \times 10^2) * (0.456 \times 10^3) = 5608.8 \approx 0.561 \times 10^4$.
- The set of representable numbers is not equispaced. In the interval $[0.1, 1]$, for example, the distance between adjacent numbers is 0.001, while in $[1, 10]$ it is 0.01.

In the example above we have seen a particular instance of a *normalized floating point number system*. A general floating point number system is a subset $\mathbb{F}$ of $\mathbb{R}$ whose nonzero elements have the form

$$x = \pm\Big(\frac{d_1}{\beta^1} + \cdots + \frac{d_t}{\beta^t}\Big) \times \beta^e. \tag{1.1}$$

Here, $\beta \in \{2, 3, \ldots\}$ is the *base* and $t \in \mathbb{N}$ the *precision* of $\mathbb{F}$. In contrast to β and t, the exponent e is not fixed but can take any integer value in the *exponent range* $e_{\min} \le e \le e_{\max}$. Similarly, each *digit* d_i can vary in the set $\{0, \ldots, \beta-1\}$. In a *normalized* floating point number system we have $d_1 \neq 0$. The number zero does not have a normalized representation. The gap between 1, which we always assume to be an element of $\mathbb{F}$, and the next largest floating point number is called *machine epsilon* $\epsilon_M = \beta^{1-t}$. The *unit roundoff* $u = \frac{1}{2}\epsilon_M$ gives a worst case bound on the relative error when approximating real numbers by floating point numbers. The following theorem summarizes some of the properties of a normalized floating point number system.

Theorem 1.1. *For $\mathbb{F} = \mathbb{F}(\beta, t, e_{\min}, e_{\max})$ the following statements are true.*

1. *Every $x \in \mathbb{F} \setminus \{0\}$ has a unique representation (1.1).*
2. *The largest and smallest positive elements of $\mathbb{F}$, respectively, are given by*
$$x_{\max} = \beta^{e_{\max}}(1 - \beta^{-t}),$$
$$x_{\min} = \beta^{e_{\min}-1}.$$
3. *The spacing between adjacent floating point numbers in the range $[\beta^{e-1}, \beta^e]$ is $\beta^{e-t} = \epsilon_M \beta^{e-1}$.*
4. *Every real number x lying in the* representable range
$$R = [-x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$$
can be approximated by an $\tilde{x} \in \mathbb{F}$ such that the relative error stays below the unit roundoff. That is,
$$|x - \tilde{x}| = \min_{y \in \mathbb{F}} |x - y| \le u|x|. \tag{1.2}$$

Proof. Exercise. □

For every floating point number system $\mathbb{F}$ we define a *rounding function* which maps every number in the representable range R to its best floating point representative:

$$\mathrm{fl} : R \to \mathbb{F}, \quad x \mapsto \mathrm{fl}(x) = \tilde{x}$$

where $\tilde{x}$ satisfies (1.2). Note that $\tilde{x}$ is not uniquely defined, if x is the exact midpoint of two adjacent floating point numbers. In this case one needs some rule deciding which one to choose. If $x \notin R$ we cannot control the roundoff error. We distinguish two cases: If $|x| > x_{\max}$, we say that x *overflows*. If $0 < |x| < x_{\min}$, then it *underflows*. The following statement is just a rephrasing of item 4 in Thm. 1.1.

Corollary 1.1. *For every $x \in R$ there is a $\delta \in [-u, u]$ such that*

$$\mathrm{fl}(x) = (1 + \delta)x. \tag{1.3}$$

Note that δ is just the relative error of $\mathrm{fl}(x)$. Equation (1.3) basically tells us what happens to a real number x when it is represented on a computer. This is the first step towards analysing the effects of rounding errors in numerical algorithms. The next step is to quantify the error committed during an arithmetic operation with floating point numbers. How large the error of a floating point operation, a *flop*, is depends on the way it is implemented on your computer. A good design principle is formulated in the following assumption, sometimes referred to as the *Fundamental Axiom of Floating Point Arithmetic:*

Assumption 1.1. Let $\circ$ stand for any of the four basic arithmetic operations $+$, $-$, $\times$ or $/$ and denote by $\circledcirc$ its floating point analogue. For all $x, y \in \mathbb{F}$ satisfying $x \circ y \in R$ we have

$$x \circledcirc y = \mathrm{fl}(x \circ y). \tag{1.4}$$

This assumption is essentially a mathematical definition of the basic floating point operations. It states that the result of any such operation (that does not lead to over- or underflow) should be the same as the rounded result of the *exact* operation. On a machine that satisfies both (1.3) and (1.4) floating point operations are exact up to a relative error of size at most u.

Normalized systems $\mathbb{F}$ can be extended by *subnormal numbers*, also called *denormalized numbers*. These nonzero numbers are characterized by having minimal exponent $e = e_{\min}$ and $d_1 = 0$. The extended floating point number system $\hat{\mathbb{F}} = \hat{\mathbb{F}}(\beta, t, e_{\min}, e_{\max})$ is thus given by

$$\hat{\mathbb{F}} = \mathbb{F} \cup \Big\{ \pm \Big(\frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \Big) \times \beta^{e_{\min}} : 0 \le d_i \le \beta - 1 \Big\}.$$

The smallest positive subnormal number is $\beta^{e_{\min} - t}$, which usually is much smaller than $x_{\min}$. Note that in contrast to normalized numbers, subnormal numbers are equidistant. Because of this, real numbers x lying in the range $0 < |x| < x_{\min}$ cannot be approximated by subnormal numbers such that the relative error stays below u.

Example 1.2 (IEEE standard)**.** Nowadays, on most computers floating point arithmetic is in conformance with the so-called IEEE 754 standard which supports two standard floating number systems: the *single format*

$$\hat{\mathbb{F}}(2, 24, -125, 128)$$

and the *double format*

$$\hat{\mathbb{F}}(2, 53, -1021, 1024).$$

This standard also satisfies Assumption 1.1 with so-called "rounding to even." That is, if the best approximation of a number $x \in \mathbb{R}$ is not unique in $\mathbb{F}$, then the one with $d_t = 0$ is chosen.

Example 1.3 (Non-associativity of floating point addition)**.** We return to our toy calculator. In accordance with Assumption 1.1 we have

$$(1000 \oplus 4) \oplus 4 = \mathrm{fl}(\mathrm{fl}(1000 + 4) + 4) = \mathrm{fl}(1000 + 4) = 1000.$$

If we change the order of addition, then

$$1000 \oplus (4 \oplus 4) = \mathrm{fl}(1000 + \mathrm{fl}(4 + 4)) = \mathrm{fl}(1000 + 8) = 1010.$$

Similar examples can be constructed involving the other three operations. From this simple observation we can already deduce the following fundamental fact:

> *In floating point arithmetic mathematically equivalent algorithms in general do not lead to the same results!*

1.3 Cancellation

Cancellation is maybe the most common reason for numerical instability. It happens when we compute the difference between two almost equal floating point numbers. Leading digits are cancelled out, while trailing digits, which are more likely to be erroneous, become significant.

Example 1.4. Consider the following computation

$$(236 + 2340) - 2560 = 2576 - 2560 = 16.$$

Performing the same computation on the 3-digit calculator we get

$$(236 \oplus 2340) \ominus 2560 = 2580 \ominus 2560 = 20,$$

which has a relative error of 25%. It is very important to realize that the reason for this is not poor accuracy of floating point subtraction. On the contrary, the subtraction is done exactly! The problem with the subtraction is that it magnifies the error of the previous addition, which was not exact. We can avoid the error amplification by simply changing the order of addition and subtraction

$$236 \oplus (2340 \ominus 2560) = 236 \oplus (-220) = 16.$$

Let us look at cancellation in a more general way. Let $\tilde{x}_1 = x_1(1+\delta_1)$ be an approximation of x_1 and $\tilde{x}_2 = x_2(1+\delta_2)$ an approximation of x_2. We would like to compute $y = x_1 - x_2$. The relative error of $\tilde{y} = \tilde{x}_1 - \tilde{x}_2$ can be bounded in the following way

$$\frac{|\tilde{y} - y|}{|y|} = \frac{|x_1\delta_1 - x_2\delta_2|}{|x_1 - x_2|} \leq \max(|\delta_1|, |\delta_2|)\frac{|x_1| + |x_2|}{|x_1 - x_2|}. \tag{1.5}$$

This inequality is sharp, that is, there are real numbers $x_1, x_2, \delta_1, \delta_2$ with $x_1 \neq x_2$ such that the inequality is an equality. Thus, in the worst case the relative error of $\tilde{y}$ is basically a large multiple of the original relative errors δ_1, δ_2. Note that the fraction on the right hand side is large, if $|x_1 - x_2|$ is much smaller than $|x_1| + |x_2|$, i.e. if x_1 and x_2 are very close in a relative sense, which is exactly when cancellation occurs.

The reasoning above also shows that cancellation is unproblematic, if we can be sure that the numbers are error-free ($\delta_1 = \delta_2 = 0$). Cancellation (with erroneous data) can also be neglected, if it does not really influence the final result of a computation. We illustrate this point with an example.

Example 1.5. We extend the previous example by an addition with a large number:

$$((236 + 2340) - 2560) + 10^8 = (2576 - 2560) + 10^8 = 16 + 10^8.$$

With 3-digit precision we get

$$((236 \oplus 2340) \ominus 2560) \oplus 10^8 = (2580 \ominus 2560) \oplus 10^8 = 10^8,$$

and the relative error is negligible.

Example 1.6 (Roots of quadratic polynomials)**.** Consider the problem of computing the roots of the quadratic polynomial $x^2 + px + q$. The standard formula

$$x_{1,2} = -p/2 \pm \sqrt{p^2/4 - q}$$

contains two possible sources of cancellation. First, if $q \approx 0$, then one of the two solutions (depending on the sign of p) is affected. This problem can be eliminated by first computing the other solution, i.e. the one which is larger in absolute value, and then using Vieta's formula $x_1x_2 = q$ to compute the smaller one. Second, cancellation can occur when $q \approx p^2/4$. This happens for polynomials with almost identical roots. This case is more severe, as there is no mathematical trick to overcome the loss of accuracy.

Unavoidable cancellations often indicate an ill-conditioned problem. Conditioning is the topic of the next section.

1.4 Condition of a Problem

The condition of a problem describes its sensitivity with respect to perturbations. A problem is *ill-conditioned*, if small perturbations of the input can lead

to drastic changes of the output. On the other hand, if all small perturbations of the input lead only to small changes in the output, the problem is *well-conditioned.* Using floating point arithmetic small perturbations are unavoidable and must be expected to occur with every arithmetic operation. Therefore, conditioning is an important concept, because it lets you identify those problems which are problematic to solve on a computer.

Mathematically, we can view a problem as a function f which must be evaluated at a certain point: $x \mapsto f(x)$. Consider the simplest case first, where f maps real numbers to real numbers. If there is a small perturbation $\tilde{x}$ of x, for example $\tilde{x} = \mathrm{fl}(x)$, such that

$$|f(\tilde{x}) - f(x)| \gg |\tilde{x} - x|, \tag{1.6}$$

we could call the problem of evaluating f at $x \in \mathbb{R}$ ill-conditioned (with respect to the absolute error). The relation $\gg$ stands for "much larger than." However, input and output of f might be of entirely different magnitudes, in which case comparing $|f(\tilde{x}) - f(x)|$ with $|\tilde{x} - x|$ is not very meaningful.

Relative errors typically are more appropriate than absolute ones in this context. Accordingly, we call the problem $x \mapsto f(x)$ *ill-conditioned* (with respect to the relative error), if there is a small perturbation $\tilde{x}$ of x such that

$$\frac{|f(\tilde{x}) - f(x)|}{|f(x)|} \gg \frac{|\tilde{x} - x|}{|x|}. \tag{1.7}$$

If the problem is not ill-conditioned, we call it *well-conditioned.* That is, for all small perturbations $\tilde{x}$, the relative error of $f(\tilde{x})$ is comparable to or smaller than the relative error of $\tilde{x}$.

Clearly, words like "small" and "large" are not mathematically precise and depend very much on the context. However, a left-hand side several orders of magnitude larger than the right-hand side often indicates an ill-conditioned problem.

For differentiable functions an obvious measure of sensitivity with respect to a change in the argument is the derivative f'. We therefore define the *absolute* and *relative condition numbers* of the problem $x \mapsto f(x)$ as

$$\kappa_{\mathrm{abs}}(x) = |f'(x)|$$

and

$$\kappa_{\mathrm{rel}}(x) = \frac{|f'(x)||x|}{|f(x)|},$$

respectively. A problem is well-conditioned, if its condition number is small. For the relative condition number this can mean that it is of order 10^2 or below.

In general we will view problems as functions $f : \mathbb{K}^n \to \mathbb{K}^m$, where we endow both $\mathbb{K}^n$ and $\mathbb{K}^m$ with certain norms $\|\cdot\|_{(n)}$ and $\|\cdot\|_{(m)}$, respectively. In this case, the absolute condition number of evaluating f at $x \in \mathbb{K}^n$ is given by

$$\kappa_{\mathrm{abs}}(x) = \|J(x)\|_{(n,m)}, \tag{1.8}$$

where

$$J = \left(\frac{\partial f_i}{\partial x_j} \right)_{ij} \in \mathbb{K}^{m \times n}$$

is the *Jacobian matrix* of f. The norm in (1.8) is the matrix norm induced by the vector norms $\|\cdot\|_{(n)}$ and $\|\cdot\|_{(m)}$. Similarly, the relative condition number is given by

$$\kappa_{\mathrm{rel}}(x) = \frac{\|J(x)\|_{(n,m)} \|x\|_{(n)}}{\|f(x)\|_{(m)}}. \tag{1.9}$$

See Sect. A.8 for more details on norms.

Example 1.7 (Division by two)**.** We consider the problem of dividing a number by two. Thus, $f(x) = x/2$ and $f'(x) = 1/2$ for all $x \in \mathbb{R}$. The relative condition number is given by

$$\kappa_{\mathrm{rel}}(x) = \frac{\frac{1}{2}|x|}{|\frac{x}{2}|} = 1,$$

which indicates a well-conditioned problem.

Example 1.8 (Cancellation)**.** Consider the function

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad x = (x_1, x_2)^\top \mapsto x_1 - x_2.$$

Its Jacobian is the 1×2 matrix $(1, -1)$. Endowing $\mathbb{R}^2$ with the 1-norm, the induced norm of J equals 1. Hence

$$\kappa_{\mathrm{rel}}(x) = \frac{\|J(x)\|_1 \|x\|_1}{|f(x)|} = \frac{|x_1| + |x_2|}{|x_1 - x_2|},$$

which is large for $x_1 \approx x_2$. We conclude that the subtraction of nearly equal numbers is ill-conditioned with respect to the relative error. Revisiting our brief analysis from (1.5) we see that the behaviour observed there is captured well by the concept of conditioning. The factor that amplified the input error $\max(\delta_1, \delta_2)$ turns out to be just the relative condition number.

Example 1.9 (Quadratic polynomials revisited)**.** Examples 1.6 and 1.8 strongly suggest that computing the roots of a polynomial with $p^2/4 \approx q$ is an ill-conditioned problem. We avoid the tedious computation of κ_{rel} in this case. Instead we only consider the polynomial $x^2 - 2x + 1 = (x-1)^2$ and show that a small perturbation of the coefficients can cause a much larger change in the roots.

Replacing the double root $x_{1,2} = 1$ of the original polynomial by $x_{1,2} = 1 \pm \Delta x$ we calculate

$$(x - 1 + \Delta x)(x - 1 - \Delta x) = x^2 - 2x + 1 - (\Delta x)^2.$$

Reading this equation backwards, we see that a perturbation of q by $(\Delta x)^2$ changes the roots by Δx. However, if $(\Delta x)^2 \ll 1$ then $|\Delta x|$ will be much greater than $(\Delta x)^2$. Note that absolute and relative conditioning are basically equivalent for this problem, because input $(p, q) = (-2, 1)$ and output $(x_1, x_2) = (1, 1)$ are of the same order.

1.5 Condition Number of a Matrix

Consider the problem of multiplying $x \in \mathbb{R}^n$ with a matrix $A \in \mathbb{R}^{m\times n}$. The Jacobian of the map $f : x \mapsto Ax$ is A itself. Thus

$$\kappa_{\text{rel}}(x) = \frac{\|A\|\|x\|}{\|Ax\|},$$

where we have dropped the subscripts in the norms for simplicity. Now assume that $m = n$ and that A is invertible. Taking the maximum over all nonzero vectors we can obtain a sharp upper bound for $\kappa_{\text{rel}}(x)$ which is independent of x

$$\frac{\|A\|\|x\|}{\|Ax\|} \leq \|A\| \max_{x\neq 0} \frac{\|x\|}{\|Ax\|} = \|A\|\|A^{-1}\|,$$

where we have used (A.9). This upper bound

$$\kappa(A) \coloneqq \|A\|\|A^{-1}\| \tag{1.10}$$

is called the *condition number of* A.

Next, consider the problem $f : b \mapsto A^{-1}b$. As before

$$\kappa_{\text{rel}}(b) = \frac{\|A^{-1}\|\|b\|}{\|A^{-1}b\|} \leq \|A^{-1}\|\|A\| = \kappa(A).$$

Thus, the condition number of A not only controls how perturbations in x affect Ax, but also how perturbations in the right hand side b affect the solution x of the linear system $Ax = b$. Finally, it can be shown that $\kappa(A)$ also captures how changes in A affect the solution x, i.e. it is an upper bound for the relative condition number of the problem $f : A \mapsto A^{-1}b$. In summary, the condition number of A controls the sensitivity of three problems: $x \mapsto b$, $b \mapsto x$ and $A \mapsto x$. Hence its central role in numerical linear algebra.

Using equation (A.8), the condition number (1.10) can be rewritten as

$$\kappa(A) = \frac{\max_{\|x\|=1} \|Ax\|}{\min_{\|x\|=1} \|Ax\|}. \tag{1.11}$$

Since the numerator of this fraction can never be smaller than the denominator, $\kappa(A) \geq 1$ must always hold. Thus, the best possible condition number any matrix can have is 1. Geometrically, expression (1.11) measures how much the transformation $x \mapsto Ax$ distorts the unit circle

$$\{x \in \mathbb{R}^n : \|x\| = 1\}.$$

And since this transformation is linear, the condition number $\kappa(A)$ effectively measures how much A distorts the whole space.

Since every induced matrix norm depends on the chosen vector norms, so does the corresponding condition number. Hence we define

$$\kappa_p(A) = \|A\|_p\|A^{-1}\|_p$$

for all $p \in [1, \infty]$.

Example 1.10 (Condition number for the spectral norm.)**.** For the spectral norm (cf. Example A.3) we get the ratio of largest to smallest singular value

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}.$$

If A is a self-adjoint matrix, then this fraction in fact equals

$$\kappa_2(A) = \frac{\max\{|\lambda| : \lambda \text{ eigenvalue of } A\}}{\min\{|\lambda| : \lambda \text{ eigenvalue of } A\}},$$

cf. the last paragraph of section A.8. If Q is an orthogonal or unitary matrix, then all its singular values equal one and $\kappa_2(Q) = 1$. Therefore, orthogonal and unitary matrices are the "best-conditioned" types of matrices. Also, multiplication of a matrix with an orthogonal or unitary matrix from the left or right does not change its condition number (cf. equation (A.12) in the appendix):

$$\kappa_2(UAV) = \|UAV\|_2 \|V^* A^{-1} U^*\|_2 = \|A\|_2 \|A^{-1}\|_2 = \kappa_2(A).$$

Example 1.11. Consider the system $Ax = b$ with matrix $A = \operatorname{diag}(1, \epsilon)$ and right hand side $b = (1, \epsilon)^\top$ for some $0 < \epsilon \ll 1$. The solution is given by $x = (1, 1)^\top$. The matrix A is self-adjoint and therefore $\kappa_2 = 1/\epsilon \gg 1$, which indicates a very ill-conditioned problem.

Indeed, a perturbation of the right-hand side with $\Delta b = (0, \epsilon)$ leads to a solution $y = A^{-1}(b + \Delta b) = (1, 2)^\top$ with relative error of about 70 percent, while the relative error of b is only ϵ. Similarly, a very small perturbation of A with $\Delta A = \operatorname{diag}(0, \epsilon)$ leads to a solution $y = (A + \Delta A)^{-1} b = (1, 0.5)^\top$ having a large relative error.

1.6 Stability of an Algorithm

In the previous two sections we have studied the sensitivity of mathematical problems with respect to perturbations. Next we turn to the actual computation of solutions using floating point arithmetic. Recall that mathematically equivalent ways of computing a certain result are not equivalent using floating point arithmetic (Examples 1.3 and 1.4), and some will lead to larger errors than others. While we have to expect roundoff errors to occur during every single operation, we want the overall error not to escalate but to stay within moderate bounds. Roughly speaking, algorithms having this robustness with respect to roundoff errors are called stable.

1.6.1 Forward and Backward Errors

Let $\tilde{y}$ be an approximation to the solution $y = f(x)$ of the problem $x \mapsto f(x)$. The absolute and relative errors of $\tilde{y}$, that is,

$$\|\tilde{y} - y\| \quad \text{and} \quad \frac{\|\tilde{y} - y\|}{\|y\|},$$

are called *forward errors* of $\tilde{y}$. Forward errors of reasonable magnitude can be hard to obtain sometimes. There is, however, another way to measure the quality of $\tilde{y}$. Suppose there is a perturbation of x, for which $\tilde{y}$ is the exact solution, i.e. $\tilde{y} = f(x + \Delta x)$. Then

$$\|\Delta x\| \quad \text{and} \quad \frac{\|\Delta x\|}{\|x\|}$$

are called the *absolute* and *relative backward errors*, respectively, of $\tilde{y}$. If there is more than one Δx satisfying $\tilde{y} = f(x + \Delta x)$, then we choose the one which is smallest in absolute value.

The condition number of f controls the relationship between forward and backward errors. For simplicity let $f : \mathbb{R} \to \mathbb{R}$ be a twice continuously differentiable function. As above denote by Δx the backward error of $\tilde{y}$. By Taylor's theorem there is a number ξ between x and $x + \Delta x$ such that

$$\tilde{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(\xi)}{2}(\Delta x)^2.$$

Dropping the quadratic term in Δx and taking absolute values we obtain

$$|\tilde{y} - y| \approx \kappa_{\text{abs}}(x)|\Delta x|.$$

Division by $|y|$ gives the corresponding relation for the relative errors

$$\frac{|\tilde{y} - y|}{|y|} \approx \kappa_{\text{rel}}(x)\frac{|\Delta x|}{|x|}.$$

We can summarize the last two relations conveniently with one rule of thumb

$$\text{FE} \lesssim \kappa \cdot \text{BE}, \tag{1.12}$$

where FE stands for (relative or absolute) forward errors, BE for backward errors, and κ is the corresponding condition number. This rule of thumb can be interpreted by looking at three different types of problems.

- If a problem is ill-conditioned ($\kappa_{\text{rel}} \gg 1$), then computed solutions $\tilde{y}$ can have a large forward error even though the backward error is small.
- On the other hand, computed solutions of an extremely well-conditioned problem ($\kappa_{\text{rel}} \ll 1$) can have small forward but very large backward errors.
- Finally, forward and backward errors of computed solutions to a well-conditioned problem ($\kappa_{\text{rel}} \approx 1$) are often comparable.

Examples 1.14, 1.15 and 1.16 below illustrate these three cases.

Example 1.12 (Cancellation)**.** Let us revisit the problem of subtracting two numbers a second time. In Section 1.3 we have shown that

$$\frac{|\tilde{y} - y|}{|y|} \leq \max(|\delta_1|, |\delta_2|)\frac{|x_1| + |x_2|}{|x_1 - x_2|}.$$

Then, in Example 1.8 we have seen that the fraction on the right hand side is nothing but the relative condition number of the problem $f : (x_1, x_2) \mapsto x_1 - x_2$. Therefore

$$\frac{|\tilde{y} - y|}{|y|} \leq \kappa_{\text{rel}}(x) \max(|\delta_1|, |\delta_2|).$$

Now, realizing that $\max(|\delta_1|, |\delta_2|)$ is essentially a relative backward error, we see that relation (1.12) is satisfied.

1.6.2 Stability

It is convenient to denote methods or algorithms for solving a problem f by $\tilde{f}$, where $\tilde{f}$ acts between the same spaces as f. If, for every x, $\tilde{f}$ produces results with a small backward error, it is called *backward stable*. Put differently, for every x, there is a small Δx such that

$$f(x + \Delta x) = \tilde{f}(x).$$

One way to interpret this definition is this: A backward stable algorithm produces the right result for almost the right problem.

A weaker notion of stability is the following. An algorithm $\tilde{f}$ is called *stable*, if for every x there are small Δx, Δy such that

$$f(x + \Delta x) = \tilde{f}(x) + \Delta y.$$

Again this can be read in a catchy way: A stable algorithm produces almost the right result for almost the right problem. Stability is weaker than backward stability in the sense that every backward stable algorithm is also stable.

As usual, the meaning of words like "small" depends on the context. However, a reasonable bound for the backward error of a backward stable algorithm is

$$\frac{\|\Delta x\|}{\|x\|} \leq Cu, \tag{1.13}$$

where $C > 0$ is a moderate constant and u is the unit roundoff of the underlying floating point number system. Since in general the input to algorithms must be assumed to be erroneous, a relative input error much smaller than u can hardly be expected. Also note that, as (1.13) is not defined if $x = 0$, what one typically tries to show is

$$\|\Delta x\| \leq Cu\|x\|,$$

implying that Δx must be zero if x is.

With essentially the same argumentation that led to (1.12), we can obtain a bound on the forward error of a backward stable algorithm

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \frac{\|f(x + \Delta x) - f(x)\|}{\|f(x)\|} \lesssim \kappa_{\text{rel}}(x) \frac{\|\Delta x\|}{\|x\|} \leq Cu\kappa_{\text{rel}}(x). \tag{1.14}$$

Since (1.14) measures the accuracy of the algorithm $\tilde{f}$, estimates of this type are sometimes called *accuracy estimates*. Roughly speaking, we can say that the accuracy of a backward stable algorithm depends on the unit roundoff of the used floating point number system, on the condition number of the underlying problem, and on the constant appearing in the backward error bound. Another way to interpret (1.14) is this: Backward stable algorithms can be expected to produce accurate results for well-conditioned problems.

1.6.3 Error Analysis

When analysing errors of a given algorithm $x \mapsto \tilde{f}(x)$ one typically replaces every floating point operation with an exact operation times a $(1+\delta)$ term, where $|\delta| \leq u$. This is in accordance with Assumption 1.1. Taylor approximations around $\delta = 0$ can then be used to simplify expressions. For $g(\delta) = 1/(1-\delta)$, for instance, first-order Taylor expansion yields

$$\frac{1}{1-\delta} \approx g(0) + g'(0)\delta = 1 + \delta.$$

Example 1.13 (Backward stability of basic floating point operations)**.** Consider the problem $f : (x_1, x_2) \mapsto x_1 + x_2$ for two real numbers x_1, x_2. Our "algorithm" in this case is $\tilde{f}(x_1, x_2) = \mathrm{fl}(x_1) \oplus \mathrm{fl}(x_2)$. Invoking (1.3) and (1.4) we compute

$$\begin{aligned}
\tilde{f}(x_1, x_2) &= \mathrm{fl}(x_1) \oplus \mathrm{fl}(x_2) \\
&= x_1(1+\delta_1) \oplus x_2(1+\delta_2) \\
&= [x_1(1+\delta_1) + x_2(1+\delta_2)](1+\delta_3) \\
&= \underbrace{x_1(1+\delta_1+\delta_3+\delta_1\delta_3)}_{\tilde{x}_1} + \underbrace{x_2(1+\delta_2+\delta_3+\delta_2\delta_3)}_{\tilde{x}_2} \\
&= f(\tilde{x}_1, \tilde{x}_2),
\end{aligned}$$

where all $|\delta_i| \leq u$. The identity $\tilde{f}(x_1, x_2) = f(\tilde{x}_1, \tilde{x}_2)$ means that the algorithm $\tilde{f}$ gives the exact answer $f(\tilde{x}_1, \tilde{x}_2)$ for a perturbed input $(\tilde{x}_1, \tilde{x}_2)$. The relative backward errors satisfy

$$\frac{|x_i - \tilde{x}_i|}{|x_i|} = |\delta_i + \delta_j + \delta_i\delta_j| \leq |\delta_i| + |\delta_j| + |\delta_i\delta_j| \approx |\delta_i| + |\delta_j| \leq 2u.$$

Since $C = 2$ is a moderate constant, floating point addition is backward stable. However, in spite of backward stability, if $x_1 \approx -x_2$, the forward errors might still be large!

Backward stability of $\ominus, \otimes, \oslash$ can be shown analogously.

Example 1.14 (An extremely well-conditioned problem)**.** Consider $f : x \mapsto 1 + x$ for $x \approx 0$. Since it does not change the outcome, we neglect rounding of

the input, i.e. we assume $x \in \mathbb{F}$, and consider the algorithm $\tilde{f}(x) = 1 \oplus x$. We have

$$\tilde{f}(x) = 1 \oplus x = (1+x)(1+\delta) = 1 + x(1+\delta+\delta/x) = f(x(1+\underbrace{\delta+\delta/x}_{\text{rel. BE}})),$$

where $|\delta| \leq u$. For very small x the fraction δ/x will be much larger than δ. Therefore, the relative backward error $\delta + \delta/x$ cannot be considered small. We conclude that $\tilde{f} : x \mapsto 1 \oplus x$ is not backward stable. (Note that this is no contradiction to Example 1.13.) However, since

$$\tilde{f}(x) = 1 + x + \delta + x\delta = f(x(1+\delta)) + \delta,$$

it is stable. The relative forward error is small as well

$$\frac{\tilde{f}(x) - f(x)}{f(x)} = \frac{\delta + x\delta}{1+x} = \delta.$$

This situation can be explained by the fact that $f : x \mapsto 1 + x$ is an extremely well-conditioned problem for x very close to zero: $\kappa_{\text{rel}}(x) = |x|/|1+x| \ll 1$. (Recall our interpretation of (1.12).) Relative errors in x are dampened heavily and thus have almost no effect on $f(x)$. This example shows that backward stability is not always a reasonable goal.

Example 1.15 (A well-conditioned problem)**.** Consider the problem $f : x \mapsto \log(1+x)$ for $x \approx 0$ which has condition number

$$\kappa_{\text{rel}}(x) = \frac{x}{(1+x)\log(1+x)} \approx \frac{1}{1+x} \approx 1,$$

since $\log(1+x) \approx x$ for x close to zero. We assume $x \in \mathbb{F}$ and that we have an exact implementation of the logarithm. Recalling that $\log(ab) = \log a + \log b$ we get

$$\begin{aligned}
\tilde{f}(x) &= \log(1 \oplus x) \\
&= \log[(1+\delta)(1+x)] \\
&= \log(1+\delta) + \log(1+x) \\
&\approx \delta + \log(1+x) \\
&= f(x)(1 + \underbrace{\delta/\log(1+x)}_{\text{rel. FE}})
\end{aligned}$$

for some $|\delta| \leq u$. The relative forward error of $\delta/\log(1+x)$ is very large for $x \approx 0$. For estimating the relative backward error ϵ we make the ansatz $\tilde{f}(x) = f(x(1+\epsilon))$, that is,

$$\log[(1+\delta)(1+x)] = \log[1 + x(1+\epsilon)].$$

Cancelling the logarithms and solving for ϵ yields $\epsilon = \delta + \delta/x$ which shows a lack of backward stability.

As we have pointed out in discussing (1.12) well-conditioned problems often lead to computed solutions with similar forward and backward errors. This is also the case for this problem, both errors are large. The issue with $x \mapsto \log(1 \oplus x)$ is very similar to cancellation. The rounding error introduced by the addition is magnified by the logarithm, even though it is evaluated exactly. Notice that the problem $y \mapsto \log y$ is ill-conditioned for $y \approx 1$.

There is, however, a mathematically equivalent rewriting

$$\log(1+x) = 2\,\mathrm{artanh}\,(x/(x+2)),$$

which avoids numerical instabilities. There are many other examples where an algebraically equivalent rewriting leads to a stable algorithm.

Example 1.16 (An ill-conditioned problem)**.** Consider $f : x \mapsto 1 - x^2$ for $|x| \approx 1$. In this case the relative condition number $\kappa_{\mathrm{rel}}(x) = 2x^2/|1-x^2|$ is very large, which indicates an ill-conditioned problem. For the sake of simplicity assume that our algorithm performs the subtraction exactly. Then

$$\tilde{f}(x) = 1 - x \otimes x = 1 - x^2(1+\delta)$$

for $\delta \leq u$ and the relative forward error explodes, since

$$\frac{\tilde{f}(x) - f(x)}{f(x)} = \frac{-x^2\delta}{1-x^2} \approx \frac{-\delta}{1-x^2}.$$

For estimating the backward error we again equate $\tilde{f}(x)$ with $f(x(1+\epsilon))$ leading to

$$1 - x^2(1+\delta) = 1 - (x(1+\epsilon))^2.$$

Solving for ϵ yields two solutions. We choose the one which is smaller in absolute value: $\epsilon = \sqrt{1+\delta} - 1$. It has a first order approximation of $\delta/2$. Thus, the relative backward error satisfies $|\epsilon| \lesssim u/2$ from which we deduce backward stability of $\tilde{f}$.

This situation is similar to the one encountered in Example 1.4. A rounding error introduced in an otherwise unproblematic computation is elevated by subsequent subtraction even though this subtraction is performed exactly.

We conclude this chapter with general advice for designing stable algorithms. Nicholas Higham, in his standard reference [8], lists the following guidelines (among others):

1. Avoid subtraction of quantities contaminated by error. Also, try to minimize the size of intermediate results relative to the size of the final result. If intermediate quantities are very large, then the final result might be affected by cancellation.

2. Look for mathematically equivalent reformulations of your problem.

3. Avoid overflow and underflow.

Chapter 2

Direct Methods for Systems of Linear Equations

This chapter's focus is on direct methods for solving systems of linear equations $Ax = b$ where $A \in \mathbb{K}^{n\times n}$ is regular. A *direct method* is an algorithm that produces a solution within a finite number of steps. In contrast, *iterative methods* in principle take infinitely many steps and have to be terminated at some point. The latter will be dealt with in the follow-up course Numerical Methods II.

Our main tool in constructing direct methods for the problem $Ax = b$ are *matrix factorizations* or *decompositions.* Their basic idea is the following:

1. Find "simple" matrices $B, C \in \mathbb{K}^{n\times n}$ such that $A = BC$.

2. Solve $By = b$ for y.

3. Solve $Cx = y$ for x.

Then, the vector x solves $Ax = b$, because $Ax = BCx = By = b$.

When using a matrix factorization approach to solve $Ax = b$, in the worst case a relative error in b will be amplified by a factor of $\kappa(B)$ to produce a new error in y. Similarly, the relative error in y can be magnified by $\kappa(C)$. In total, a relative error in b propagates to the final result x with an amplifying factor of $\kappa(B)\kappa(C)$. Thus, solving $Ax = b$ via the factorization $A = BC$ is only a good idea, if the product of condition numbers $\kappa(B)\kappa(C)$ is not much larger than $\kappa(A)$.

A general rule in numerical analysis says that one should avoid explicit computation of A^{-1}. First, it is superfluous, if only $x = A^{-1}b$ is needed. Another important reason is that computation of A^{-1} is slower and less stable than Gaussian elimination, for instance. In addition, many matrices arising in practice are very large but *sparse*, meaning that most of their entries are zero. The inverse of a sparse matrix, however, is not sparse in general and would require a large amount of space to store.

The structure and presentation of the topics covered in this chapter follows, more or less closely, the corresponding lectures in [16].

2.1 Triangular Systems

2.1.1 Forward Substitution

Let $L = (\ell_{ij}) \in \mathbb{K}^{n\times n}$ be a regular lower triangular matrix, that is, $\ell_{ij} = 0$ for $i < j$. For a given right hand side $b \in \mathbb{K}^n$ we consider the system of linear equations $Lx = b$, which takes the form

$$\begin{array}{ccccccc} \ell_{11}x_1 & & & & & = & b_1 \\ \ell_{21}x_1 & + & \ell_{22}x_2 & & & = & b_2 \\ \vdots & \vdots & & \ddots & & \vdots & \vdots \\ \ell_{n1}x_1 & + & \cdots & + & \ell_{nn}x_n & = & b_n. \end{array}$$

There is an obvious procedure for solving such a system in n steps. We start by solving the first equation for x_1 yielding

$$x_1 = b_1/\ell_{11}. \tag{2.1}$$

Next, we plug this solution into the second equation and obtain x_2. More generally, in the k-th step we plug the $k-1$ previously obtained unknowns $x_1, \ldots, x_{k-1}$ into the k-th equation an solve for x_k. In fact, x_k can be expressed explicitly in terms of $x_1, \ldots, x_{k-1}$ in a simple way

$$x_k = \Big(b_k - \sum_{j=1}^{k-1} \ell_{kj}x_j\Big)/\ell_{kk}, \quad k = 2, \ldots, n. \tag{2.2}$$

The procedure of calculating x according to (2.1) and (2.2) is called *forward substitution.* Using Matlab syntax it can be written in the following way.

Algorithm 2.1: Forward Substitution

```
x(1) = b(1)/L(1,1);
for k = 2:n
    x(k) = (b(k) - L(k,1:k-1)*x(1:k-1))/L(k,k);
end
```

For every algorithm we will try to answer the following three questions:

- Does it succeed for all inputs? If not, when does it succeed and when does it fail?
- How many flops[1] (floating point operations) does it require? Here we are mainly interested in asymptotic operation counts as $n \to \infty$, where n is the size of the input. For algorithms requiring a polynomial number of flops this means that all lower order terms are discarded, since for large n only the highest order term is significant.

[1] There is an obvious difference between flops with real numbers and flops involving complex numbers. In these notes, however, we will neglect this distinction.

- Is it (backward) stable?

Theorem 2.1. *If $L \in \mathbb{K}^{n\times n}$ is a regular lower triangular matrix, then for every $b \in \mathbb{K}^n$ forward substitution, as given by (2.1) and (2.2), yields the unique solution of $Lx = b$.*

Proof. The procedure outlined above could only break down, if $\ell_{kk} = 0$ for some k. However, a triangular matrix is regular, if and only if all its diagonal entries are nonzero. Recall that for every (upper or lower) triangular matrix the determinant equals the product of its diagonal entries. Therefore, if L is invertible, forward substitution cannot fail. □

Theorem 2.2. *For $L \in \mathbb{K}^{n\times n}$ and $b \in \mathbb{K}^n$ forward substitution requires n^2 flops.*

Proof. According to (2.2) the k-th step of forward substitution consists of one division, $k-1$ multiplications and $k-1$ subtractions, in total $1+2(k-1) = 2k-1$ operations. Thus, the number of flops involved in the whole procedure equals

$$\sum_{k=1}^{n}(2k-1) = 2\frac{n(n+1)}{2} - n = n^2.$$

In the summation above we used the fact that $\sum_{k=1}^{n} k = n(n+1)/2$. □

Is forward substitution backward stable? That is, can we assert that an algorithm implementing (2.2) produces results $\tilde{x}$ solving a linear system which is an only slightly perturbed version of the exact problem $Lx = b$? The following theorem gives a positive answer.

Theorem 2.3. *Forward substitution is backward stable. That is, for all regular lower triangular matrices $L \in \mathbb{F}^{n\times n}$ and $b \in \mathbb{F}^n$ forward substitution implemented on a machine satisfying Corollary 1.1 and Assumption 1.1 will produce a result $\tilde{x} \in \mathbb{F}^n$ which solves a perturbed system $\tilde{L}\tilde{x} = b$ with componentwise errors*

$$\frac{|\ell_{ij} - \tilde{\ell}_{ij}|}{|\ell_{ij}|} \leq nu + \text{h.o.t.}, \tag{2.3}$$

if $\ell_{ij} \neq 0$. If $\ell_{ij} = 0$, then also $\tilde{\ell}_{ij} = 0$.

A few remarks are in order before we prove this statement. First, the u in the estimate above is the unit roundoff as introduced in the previous chapter and h.o.t. stands for higher order terms in u. Second, note that $\tilde{x}$ is asserted to solve a system with *exact* right hand side b. Strictly speaking, this is not required for establishing backward stability of forward substitution. However, if it can be shown that the backward error in one of the inputs actually equals 0, it does make sense to make the additional effort.

Proof. We give a detailed proof only for the case $n = 3$. This should convince the reader of the theorem's validity for arbitrary $n \in \mathbb{N}$.

The first component of $\tilde{x}$ is given by

$$\tilde{x}_1 = b_1 \oslash \ell_{11} = \frac{b_1}{\ell_{11}}(1+\epsilon),$$

where ϵ is a number which, according to Assumption 1.1, satisfies $|\epsilon| \leq u$. The way the theorem is stated we want to interpret all appearing errors as perturbations in L. (b is exact!) In the case of $\tilde{x}_1$ this means that we have to move the error term $1+\epsilon$ to the denominator. If we define $\epsilon' = -\epsilon/(1+\epsilon)$, then $1+\epsilon' = 1/(1+\epsilon)$ and we get

$$\tilde{x}_1 = \frac{b_1}{\ell_{11}(1+\epsilon')}.$$

Taylor expansion of ϵ' (as a function of ϵ) around zero shows that $\epsilon' = -\epsilon$ plus higher order terms. Therefore $|\epsilon'| \leq u + \text{h.o.t.}$ Note how this ϵ'-trick allows us to shift error terms from numerator and denominator and vice versa.

For the second unknown we get

$$\tilde{x}_2 = (b_2 \ominus \ell_{21} \otimes \tilde{x}_1) \oslash \ell_{22} = \frac{[b_2 - \ell_{21}\tilde{x}_1(1+\epsilon_1)]\,(1+\epsilon_2)}{\ell_{22}}(1+\epsilon_3),$$

where again $|\epsilon_i| \leq u$ for $i = 1, 2, 3$. Defining ϵ_2' and ϵ_3' as above, we can interpret all three error terms as perturbations in matrix coefficients

$$\tilde{x}_2 = \frac{b_1 - \ell_{21}\tilde{x}_1(1+\epsilon_1)}{\ell_{22}(1+\epsilon_2')(1+\epsilon_3')}.$$

The third component is slightly more involved. If we subtract from left to right (recall that order matters), we have

$$\begin{aligned}\tilde{x}_3 &= ((b_3 \ominus \ell_{31} \otimes \tilde{x}_1) \ominus \ell_{32} \otimes \tilde{x}_2) \oslash \ell_{33} \\ &= \frac{[(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4))\,(1+\epsilon_5) - \ell_{32}\tilde{x}_2(1+\epsilon_6)]\,(1+\epsilon_7)}{\ell_{33}}(1+\epsilon_8).\end{aligned}$$

The errors ϵ_4 and ϵ_6 can already be interpreted as perturbations in ℓ_{31} and ℓ_{32}, respectively. Employing the ϵ'-trick we can move the terms $1+\epsilon_7$ and $1+\epsilon_8$ into the denominator yielding

$$\tilde{x}_3 = \frac{(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4))\,(1+\epsilon_5) - \ell_{32}\tilde{x}_2(1+\epsilon_6)}{\ell_{33}(1+\epsilon_7')(1+\epsilon_8')}.$$

It only remains to take care of ϵ_5. We divide numerator and denominator by $1+\epsilon_5$ and switch to $1+\epsilon_5'$ twice:

$$\tilde{x}_3 = \frac{(b_3 - \ell_{31}\tilde{x}_1(1+\epsilon_4)) - \ell_{32}\tilde{x}_2(1+\epsilon_5')(1+\epsilon_6)}{\ell_{33}(1+\epsilon_5')(1+\epsilon_7')(1+\epsilon_8')}.$$

If we make the following definitions

$$\begin{aligned}
\tilde{\ell}_{11} &= \ell_{11}(1+\epsilon'),\\
\tilde{\ell}_{21} &= \ell_{21}(1+\epsilon_1),\\
\tilde{\ell}_{22} &= \ell_{22}(1+\epsilon'_2)(1+\epsilon'_3),\\
\tilde{\ell}_{31} &= \ell_{31}(1+\epsilon_4),\\
\tilde{\ell}_{32} &= \ell_{32}(1+\epsilon'_5)(1+\epsilon_6),\\
\tilde{\ell}_{33} &= \ell_{33}(1+\epsilon'_5)(1+\epsilon'_7)(1+\epsilon'_8),
\end{aligned}$$

what we have shown so far is that $\tilde{L}\tilde{x} = b$, where $\tilde{L}$ is the lower triangular matrix with entries $\tilde{\ell}_{ij}$. In other words, the computed solution $\tilde{x}$ solves a system with exact right hand side b and perturbed matrix.

It remains to estimate the backward errors. If $\ell_{ij} = 0$, then clearly $\tilde{\ell}_{ij} = 0$ as well. If not, then let $\delta\ell_{ij} = (\ell_{ij} - \tilde{\ell}_{ij})/\ell_{ij}$ denote the relative componentwise errors of the perturbed matrix. From the definitions of the $\tilde{\ell}_{ij}$ above it follows directly that

$$\begin{bmatrix} |\delta\ell_{11}| & & \\ |\delta\ell_{21}| & |\delta\ell_{22}| & \\ |\delta\ell_{31}| & |\delta\ell_{32}| & |\delta\ell_{33}| \end{bmatrix} \leq u \begin{bmatrix} 1 & & \\ 1 & 2 & \\ 1 & 2 & 3 \end{bmatrix} + \text{h.o.t.}$$

Since the largest error occurs for ℓ_{33}, we have shown that

$$|\delta\ell_{ij}| \leq |\delta\ell_{33}| \leq 3u + \text{h.o.t},$$

which proves the claim for $n = 3$. □

Remark 2.1. Recall the accuracy estimate (1.14) for backward stable algorithms. Theorem 2.3 now tells us that, for well-conditioned matrices L, we can expect floating point implementations of forward substitution to produce accurate results.

Remark 2.2. Note that Theorem 2.3 assumes the entries of L and b to be floating point numbers. It can be adapted to the more general case when $L \in \mathbb{K}^{n\times n}$ and $b \in \mathbb{K}^n$.

2.1.2 Back Substitution

For an upper triangular matrix $U \in \mathbb{K}^{n\times n}$, $u_{ij} = 0$ for $i > j$, the system $Ux = b$ takes the form

$$\begin{array}{ccccccccc}
u_{11}x_1 & + & \cdots & + & \cdots & + & u_{1n}x_n & = & b_1 \\
 & & u_{22}x_2 & + & \cdots & + & u_{2n}x_n & = & b_2 \\
 & & & & \ddots & & \vdots & \vdots & \vdots \\
 & & & & & & u_{nn}x_n & = & b_n.
\end{array}$$

There is a procedure for solving such systems which is completely analogous to forward substitution. The only difference is that you now start with the last equation and iterate through all equations from bottom to top. Accordingly, it is called *back substitution*. Theorems 2.1, 2.2 and 2.3 have obvious analogues for back substitution.

2.2 LU Factorization

Given a regular matrix A the aim of LU factorization is to find a lower triangular matrix L and an upper triangular matrix U such that $A = LU$. Having found such a decomposition, the system $Ax = b$ can be solved in two simple steps: first forward substitution for $Ly = b$ and then back substitution for $Ux = y$.

2.2.1 Gaussian Elimination

LU factorizations can be found using a well-known algorithm from linear algebra: *Gaussian elimination*. Gaussian elimination is the process of transforming a given matrix A into an upper triangular one U by using row operations. For an $n \times n$ matrix it consists of $n-1$ steps. In the k-th step ($k = 1, \ldots, n-1$) one eliminates all nonzero entries in the k-th column below the main diagonal. The connection to LU factorizations becomes apparent when interpreting the row operations as matrix multiplications: Each of the $n-1$ steps can be realized by multiplication from the left with a lower triangular matrix L_k.

$$\underset{A}{\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}} \xrightarrow{L_1} \underset{L_1A}{\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix}} \xrightarrow{L_2} \underset{L_2L_1A}{\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & * & * \end{bmatrix}} \xrightarrow{L_3} \underset{L_3L_2L_1A=U}{\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & & * & * \\ & & & * \end{bmatrix}}.$$

Example 2.1. Consider the following matrix[2]

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

For a 4×4 matrix Gaussian elimination consists of three steps. First, we subtract two times the first row from the second, four times the first row from the third, and three times the first row from the fourth. These row operations can be realized by multiplying A from the left with the matrix

$$L_1 = \begin{bmatrix} 1 & & & \\ -2 & 1 & & \\ -4 & & 1 & \\ -3 & & & 1 \end{bmatrix}.$$

[2] This example is borrowed from [16].

Thus the resulting matrix is given by

$$L_1 A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 3 & 5 & 5 \\ & 4 & 6 & 8 \end{bmatrix}.$$

We proceed to the second column and subtract three times the second row from the third and four times the second row from the fourth. Using matrix multiplications we can write the result as

$$L_2 L_1 A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -3 & 1 & \\ & -4 & & 1 \end{bmatrix} L_1 A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & 2 & 4 \end{bmatrix}.$$

Now, after subtracting the third row from the fourth we obtain an upper triangular matrix U.

$$L_3 L_2 L_1 A = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -1 & 1 \end{bmatrix} L_2 L_1 A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix} = U.$$

By inverting the matrices L_k we can write A as

$$A = (L_3 L_2 L_1)^{-1} U = L_1^{-1} L_2^{-1} L_3^{-1} U.$$

If the product $L_1^{-1} L_2^{-1} L_3^{-1}$ happens to be lower triangular, then we have found an LU factorization of A. The inverses are given by

$$L_1^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}, \quad L_2^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & 3 & 1 & \\ & 4 & & 1 \end{bmatrix}, \quad L_3^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1 & 1 \end{bmatrix}.$$

And their product is indeed lower triangular

$$L = L_1^{-1} L_2^{-1} L_3^{-1} = \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}.$$

Summarizing, we have found the following factorization of A

$$\underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & & 2 & 2 \\ & & & 2 \end{bmatrix}}_{U}.$$

General Formulas

Two observations are in order concerning the previous example. First, the inverses L_k^{-1} differ from L_k only in the signs of their subdiagonal entries. Second, the product $L_1^{-1}L_2^{-1}L_3^{-1}$ has a very simple structure as well: It collects the subdiagonal entries of all the L_k^{-1} into one identity matrix. Consequently, $L_1^{-1}L_2^{-1}L_3^{-1}$ is not only lower triangular but is also very easily determined from L_1, L_2 and L_3. We show below that calculation of L is *always* that easy.

Let $A \in \mathbb{K}^{n\times n}$ be a regular matrix and denote by $A^{(k)}$ be the system matrix at the beginning of step k of Gaussian elimination. That is,

$$\begin{aligned} A^{(1)} &= A \\ A^{(k)} &= L_{k-1}A^{(k-1)} = L_{k-1}\cdots L_1 A, \quad k = 2,\ldots,n-1. \end{aligned}$$

Next, for every k define the numbers

$$\ell_{jk} = \frac{a_{jk}^{(k)}}{a_{kk}^{(k)}}, \quad j = k+1,\ldots,n.$$

Then, the row operation matrix L_k is given by

$$L_k = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\ell_{k+1,k} & 1 & & \\ & & \vdots & & \ddots & \\ & & -\ell_{n,k} & & & 1 \end{bmatrix}.$$

If we define the vector

$$\ell_k = (0,\ldots,0,\ell_{k+1,k},\ldots,\ell_{n,k})^\top \in \mathbb{K}^n,$$

we can write this matrix as

$$L_k = I - \ell_k e_k^*, \tag{2.4}$$

where e_k is the k-th canonical basis vector. With these definitions at hand we can prove in full generality the two observations made in the previous paragraph.

Lemma 2.1. *The inverse of L_k is given by*

$$L_k^{-1} = I + \ell_k e_k^*.$$

Proof. We have to show that $L_k(I + \ell_k e_k^*) = (I + \ell_k e_k^*)L_k = I$. Using (2.4) we compute

$$(I - \ell_k e_k^*)(I + \ell_k e_k^*) = (I + \ell_k e_k^*)(I - \ell_k e_k^*) = I - \ell_k e_k^* \ell_k e_k^*.$$

But $e_k^*\ell_k = 0$, since the k-th entry of ℓ_k is zero, and therefore $\ell_k e_k^*\ell_k e_k^* = 0$. □

Lemma 2.2. *The product* $L_k^{-1}L_{k+1}^{-1}$ *equals* $I + \ell_k e_k^* + \ell_{k+1}e_{k+1}^*$.

Proof. We use the previous lemma and calculate

$$L_k^{-1}L_{k+1}^{-1} = (I + \ell_k e_k^*)(I + \ell_{k+1}e_{k+1}^*) = I + \ell_k e_k^* + \ell_{k+1}e_{k+1}^* + \ell_k e_k^*\ell_{k+1}e_{k+1}^*.$$

As before, the last term vanishes, because $e_k^*\ell_{k+1} = 0$. □

For the product of all $n-1$ matrices we get with essentially the same argumentation

$$L_1^{-1}\cdots L_{n-1}^{-1} = I + \sum_{k=1}^{n-1}(\ell_k e_k^*) = \begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{bmatrix} = L.$$

Finally, recall that U is just the result of successive row operations, which can be written as

$$U = L_{n-1}\cdots L_1 A.$$

Algorithm

Having found general formulas for L and U we can write down an algorithm that, for a given matrix A, computes these factors. The algorithm's title will become clear in section 2.2.2.

Algorithm 2.2: Gaussian Elimination without Pivoting

```
1 L = eye(n);
2 U = A;
3 for k = 1:n-1
4     for j = k+1:n
5         L(j,k) = U(j,k)/U(k,k);
6         U(j,k:n) = U(j,k:n) - L(j,k)*U(k,k:n);
7     end
8 end
```

Note how the matrices L_k are never formed explicitly. The row operations can be performed without them and the values ℓ_{jk} are written directly into an identity matrix. (The Matlab command `eye(n)` creates an $n \times n$ identity matrix.) Actually, not even explicit forming of L and U is necessary, as their entries could be stored into A.

How many flops does Algorithm 2.2 require? There are three for-loops: two obvious ones plus a hidden one in line 6 where two vectors are subtracted. Therefore, we should expect a figure which is cubic in n. The following theorem supports this claim.

Theorem 2.4. *To leading order Gaussian elimination requires* $\frac{2}{3}n^3$ *operations.*

Proof. The vector `U(j,k:n)` has $n-k+1$ entries. Thus, there are $2n-2k+3$ arithmetic operations ($n-k+1$ multiplications, $n-k+1$ subtractions, and one division) at each step of the inner loop. The total number of operations required equals

$$\sum_{k=1}^{n-1}\sum_{j=k+1}^{n}(2n-2k+3) \approx \sum_{k=1}^{n-1}\sum_{j=k+1}^{n} 2(n-k).$$

Since $n-k$ does not depend on the inner summation index j and the inner sum has $n-(k+1)+1 = n-k$ summands, we have

$$\sum_{k=1}^{n-1}\sum_{j=k+1}^{n} 2(n-k) = \sum_{k=1}^{n-1} 2(n-k)^2 = 2\sum_{k=1}^{n-1} k^2.$$

For evaluating this sum we use the formula

$$\sum_{i=1}^{m} i^2 = \frac{1}{6}m(m+1)(2m+1),$$

and obtain

$$2\sum_{k=1}^{n-1} k^2 = \frac{2}{6}(n-1)n(2n-1) \approx \frac{2}{3}n^3,$$

where we have dropped all lower order terms. □

Remark 2.3. Recall that forward substitution, and for reasons of symmetry also backward substitution, requires only n^2 operations. Thus, the work involved in solving $Ax = b$ via LU factorization and subsequent forward and back substitution is dominated by the factorization step.

Existence and Uniqueness

There are some basic questions about LU decomposition that we have not addressed so far: Can every regular matrix be decomposed into a product LU? If not, which matrices can and which cannot? Assuming that an LU factorization exists, is it unique?

Clearly, Algorithm 2.2 is not well-defined for every regular matrix A. If, for instance, $a_{11} = 0$, then in line 5 a division by zero will occur. Or, more generally, if for any k the diagonal entry `U(k,k)` turns out to be zero, then the algorithm will break down. However, `U(k,k)` being zero means that the $k \times k$ *leading principal submatrix*

$$(a_{ij})_{i,j=1}^{k} = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{bmatrix} \tag{2.5}$$

of the original matrix A is singular. Why? Because a sequence of row operations have produced a row consisting of all zeros. If, on the other hand, all submatrices $(a_{ij})_{i,j=1}^{k}$ are regular, then all `U(k,k)` will be nonzero. Hence the algorithm does not fail, but instead produces two matrices L and U as desired. Thus we have found a condition which implies existence of an LU factorization. In fact, this condition can be shown to be *equivalent* to the existence of an LU factorization.

Theorem 2.5. *A regular matrix $A \in \mathbb{K}^{n\times n}$ can be written $A = LU$ with $L \in \mathbb{K}^{n\times n}$ lower triangular and $U \in \mathbb{K}^{n\times n}$ upper triangular, if and only if all leading principal submatrices are regular. In this case there is a unique* normalized LU factorization*, that is, $\ell_{jj} = 1$ for $j = 1, \ldots, n$.*

This theorem can also be interpreted in the following way: The matrices for which Algorithm 2.2 succeeds are exactly those which possess an LU factorization.

Below we give two types of matrices which by Thm. 2.5 always admit an LU factorization.

Example 2.2. A matrix $A \in \mathbb{K}^{n\times n}$ is an *SPD matrix*, if it is self-adjoint and positive definite. That is, $A^* = A$ and $x^*Ax > 0$ for all nonzero $x \in \mathbb{K}^n$. Such matrices are always regular, because $x^*Ax > 0$ implies $Ax \neq 0$ for all $x \neq 0$ and therefore $\ker A = \{0\}$. In addition, every submatrix $(a_{ij})_{i,j=1}^{k}$ of an SPD matrix is again SPD. Thus, Theorem 2.5 applies.

Example 2.3. A matrix $A \in \mathbb{R}^{n\times n}$ is *strictly diagonally dominant*, if

$$|a_{jj}| > \sum_{k\neq j} |a_{jk}|$$

for all $j = 1, \ldots, n$. Such matrices are again always regular. Every submatrix $(a_{ij})_{i,j=1}^{k}$ of a strictly diagonally dominant matrix is again strictly diagonally dominant, hence regular.

Instability of Gaussian Elimination

Interestingly, nonexistence of an LU factorization is related to instability of Gaussian elimination. This connection can be illustrated conveniently by means of an example.

Example 2.4. Consider the regular matrix

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

We have pointed out already that, since $a_{11} = 0$, Algorithm 2.2 cannot be used to compute an LU factorization of A. Moreover, Theorem 2.5 tells us that this is not the algorithm's fault, because A does not have an LU factorization.

Now replace the zero by a very small number $\epsilon > 0$. Then the resulting matrix A_ϵ does have an LU factorization. It is given by

$$\underbrace{\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}}_{A_\epsilon} = \underbrace{\begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} \epsilon & 1 \\ 0 & 1-\epsilon^{-1} \end{bmatrix}}_{U}$$

For the sake of simplicity, assume that ϵ^{-1} so large that $1 - \epsilon^{-1}$ cannot be represented exactly on our machine and that $\mathrm{fl}(1-\epsilon^{-1})$ happens to be equal to $-\epsilon^{-1}$. Consequently, running Algorithm 2.2 on this machine will produce the factors

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ \epsilon^{-1} & 1 \end{bmatrix}, \quad \tilde{U} = \begin{bmatrix} \epsilon & 1 \\ 0 & -\epsilon^{-1} \end{bmatrix}.$$

While the errors in $\tilde{L}$ and $\tilde{U}$ might seem acceptable, their product

$$\tilde{A}_\epsilon \coloneqq \tilde{L}\tilde{U} = \begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix}$$

is entirely different from A_ϵ. In other words, the forward error is small, but the backward error is not. Therefore a solution of $\tilde{L}\tilde{U}x = b$ will in general differ considerably from a solution of $A_\epsilon x = b$. Choosing $b = (1,0)^\top$, for instance, we get $(\tilde{L}\tilde{U})^{-1}b = (0,1)^\top$, but the actual solution is $A_\epsilon^{-1}b \approx (-1,1)^\top$. The underlying problem is that the factorization of A_ϵ was *not* backward stable.

Why? Using the notation of Section 1.6 what we wanted to find is the exact factorization $f(A_\epsilon) = (L,U)$. Instead we obtained $\tilde{f}(A_\epsilon) = (\tilde{L},\tilde{U})$, where $\tilde{L}\tilde{U} = \tilde{A}_\epsilon \neq A_\epsilon$. The relative backward error of this computed factorization is far from small. Measured in the ∞-norm, for instance, it amounts to

$$\frac{\|\tilde{A}_\epsilon - A_\epsilon\|_\infty}{\|A_\epsilon\|_\infty} = \frac{1}{2}.$$

The actual issue, however, is not so much the algorithm we used to find the factorization, but the factorization itself. At the very beginning of Chapter 2 on page 17 we argued that a matrix factorization approach $A = BC$ is only reasonable if $\kappa(A) \approx \kappa(B)\kappa(C)$. However, this rule of thumb can be severely violated by the LU factorization. In particular it is violated for the normalized LU factorization of A_ϵ, since $\kappa_\infty(A_\epsilon) \approx 4$ while $\kappa_\infty(L) \approx \kappa_\infty(U) \approx \epsilon^{-2}$.

2.2.2 Pivoting

There is a surprisingly simple remedy to the situation encountered in Example 2.4. Let us try to pinpoint what exactly caused the large condition numbers. Applying Gaussian elimination to A_ϵ the first thing to do is to compute

$$\ell_{21} = \frac{a_{\epsilon,21}}{a_{\epsilon,11}} = \frac{1}{\epsilon}.$$

This number turned out to be huge and directly affected $\|L\|_\infty$ and $\kappa_\infty(L)$. However, this can be avoided by simply interchanging the rows of A_ϵ first. Recall that, when solving a linear system $Ax = b$, interchanging rows is fine as long as we interchange the corresponding entries of b as well. So, if we apply the elimination step to

$$PA_\epsilon = \begin{bmatrix} & 1 \\ 1 & \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix},$$

instead of A_ϵ, then $\ell_{21} = \epsilon$ and $\|L\|_\infty \approx 1$. This is the basic idea of pivoting.

In step k of Gaussian elimination, we use the (k, k) element of the system matrix $a_{kk}^{(k)}$ to introduce zeros in the k-th column

$$\begin{bmatrix} * & * & * & * \\ & a_{kk}^{(k)} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{kk}^{(k)} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix}.$$

This element is usually called the *pivot*. We have seen above that this elimination is impossible, if the pivot is zero, or problematic, if it is much smaller in absolute value than one of the entries below. However, instead of $a_{kk}^{(k)}$ we could equally use any element $a_{ik}^{(k)}, i \geq k$, to introduce zeros in column k. Ideally we would like to choose that element $a_{ik}^{(k)}$ which is largest in absolute value. In addition, in order to keep the triangular structure we can interchange rows so that $a_{ik}^{(k)}$ moves into the main diagonal. For example,

$$\begin{bmatrix} * & * & * & * \\ & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & * \\ & a_{ik}^{(k)} & * & * \\ & 0 & * & * \\ & 0 & * & * \end{bmatrix}.$$

This strategy is called *partial pivoting*. Using matrix products we can write the result of $n - 1$ steps of Gaussian elimination with partial pivoting in the following way.

$$L_{n-1}P_{n-1} \cdots L_1 P_1 A = U,$$

where the P_j are permutation matrices.

More generally, we could look not only at the k-th column for the largest element but at all entries $a_{ij}^{(k)}$, where both $i \geq k$ and $j \geq k$, and then interchange rows and columns in order to move it into the (k, k) position. This more expensive strategy is called *complete pivoting*. In practice, however, partial pivoting is often sufficient.

Example 2.5. We consider the same 4×4 matrix as before

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

and apply Gaussian elimination with partial pivoting. The largest element in the first column is a_{31}. Therefore we interchange rows three and one. This is achieved by multiplying A from the left with an identity matrix that has rows one and three interchanged

$$P_1 A = \begin{bmatrix} & & 1 & \\ & 1 & & \\ 1 & & & \\ & & & 1 \end{bmatrix} A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix}.$$

Next we eliminate

$$L_1 P_1 A = \begin{bmatrix} 1 & & & \\ -\frac{1}{2} & 1 & & \\ -\frac{1}{4} & & 1 & \\ -\frac{3}{4} & & & 1 \end{bmatrix} P_1 A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \end{bmatrix}.$$

Since $\frac{7}{4}$ is greater than both $-\frac{3}{4}$ and $-\frac{1}{2}$ in absolute value, we interchange rows two and four before eliminating

$$\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix}}_{L_2} \underbrace{\begin{bmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{bmatrix}}_{P_2} L_1 P_1 A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{2}{7} & \frac{4}{7} \\ & & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix}.$$

Finally, to obtain an upper triangular matrix we interchange the third and fourth rows and then introduce a zero in position $(4, 3)$.

$$\underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{1}{3} & 1 \end{bmatrix}}_{L_3} \underbrace{\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix}}_{P_3} L_2 P_2 L_1 P_1 A = \begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix} = U.$$

In the end we have obtained $L_3 P_3 L_2 P_2 L_1 P_1 A = U$. Now, the natural question is whether this gives us an LU factorization of A. In short the answer is no, because the product $L_3 P_3 L_2 P_2 L_1 P_1$ in general is far from lower triangular, and so is its inverse. However, we can rearrange the matrix product into

$$L_3 P_3 L_2 P_2 L_1 P_1 = L_3' L_2' L_1' P_3 P_2 P_1,$$

where L_j' is equal to L_j up to a permutation of subdiagonal entries. More precisely, the L_j' are given by

$$L_3' = L_3, \quad L_2' = P_3 L_2 P_3^{-1}, \quad L_1' = P_3 P_2 L_1 P_2^{-1} P_3^{-1}.$$

For example,

$$L_2' = P_3 L_2 P_3^{-1} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{3}{7} & 1 & \\ & \frac{2}{7} & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{bmatrix}$$
$$= \begin{bmatrix} 1 & & & \\ & 1 & & \\ & \frac{2}{7} & 1 & \\ & \frac{3}{7} & & 1 \end{bmatrix}.$$

Note that the product of lower triangular matrices $L_3' L_2' L_1'$ is again lower triangular and so is their inverse. Similarly the product of permutation matrices is again a permutation matrix. We therefore define $L \coloneqq (L_3' L_2' L_1')^{-1}$ and $P \coloneqq P_3 P_2 P_1$ and return to the factorization of A

$$U = L_3 P_3 L_2 P_2 L_1 P_1 A = L_3' L_2' L_1' P_3 P_2 P_1 A = L^{-1} P A,$$

or equivalently

$$\underbrace{\begin{bmatrix} & & 1 & \\ & & & 1 \\ & 1 & & \\ 1 & & & \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & & & \\ \frac{3}{4} & 1 & & \\ \frac{1}{2} & -\frac{2}{7} & 1 & \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 8 & 7 & 9 & 5 \\ & \frac{7}{4} & \frac{9}{4} & \frac{17}{4} \\ & & -\frac{6}{7} & -\frac{2}{7} \\ & & & \frac{2}{3} \end{bmatrix}}_{U}$$

Thus, we have found an LU factorization not of A but of PA, which is a row-permuted version of A. Besides, note how all subdiagonal entries of L are ≤ 1 in absolute value because of the pivoting strategy.

General Formulas and Algorithm

Let A be a regular $n \times n$ matrix. Then, after $n-1$ steps of Gaussian elimination with partial pivoting we can write the resulting upper triangular matrix U in the following way

$$U = L_{n-1} P_{n-1} \cdots L_1 P_1 A.$$

Defining the modified row operation matrices as

$$L_k' = P_{n-1} \cdots P_{k+1} L_k P_{k+1}^{-1} \cdots P_{n-1}^{-1} \tag{2.6}$$

for $k = 1, \ldots, n-1$, we can rewrite the above expression for U

$$U = \underbrace{L_{n-1}' \cdots L_1'}_{=L^{-1}} \underbrace{P_{n-1} \cdots P_1}_{=P} A.$$

Note that the L'_k have the same structure as the L_k, since

$$\begin{aligned} L'_k &= P_{n-1}\cdots P_{k+1}(I-\ell_k e_k^*)P_{k+1}^{-1}\cdots P_{n-1}^{-1} \\ &= I - \underbrace{P_{n-1}\cdots P_{k+1}\ell_k}_{\ell'_k}\underbrace{e_k^* P_{k+1}^{-1}\cdots P_{n-1}^{-1}}_{(P_{n-1}\cdots P_{k+1}e_k)^*} \\ &= I - \ell'_k e_k^*, \end{aligned}$$

where we have used the facts that, first, every permutation matrix is orthogonal and, second, that the k-th permutation matrix P_k leaves the first $k-1$ entries unchanged. Therefore, inverting and multiplying the L'_k is as easy as before, recall Lemmas 2.1 and 2.2

We are now in a position to write down the algorithm. For the sake of readability we use pseudocode instead of correct Matlab syntax.

Algorithm 2.3: Gaussian Elimination with Partial Pivoting

```
L = I;
U = A;
P = I;
for k = 1:n-1
    select i >= k to maximize |U(i,k)|
    interchange U(k,k:n) and U(i,k:n)
    interchange L(k,1:k-1) and L(i,1:k-1)
    interchange P(k,:) and P(i,:)
    for j = k+1:n
        L(j,k) = U(j,k)/U(k,k);
        U(j,k:n) = U(j,k:n) - L(j,k)*U(k,k:n);
    end
end
```

A few remarks are in order regarding actual implementation of this algorithm. In the first iteration of the outer `for`-loop, i.e. when `k = 1`, no entries of `L` should be interchanged. Compare with equation (2.6), where only P_k with $k \geq 2$ appear. Second, there is no need to represent P as a matrix. The same can be achieved more efficiently using a permutation vector, that is, a vector which is initialized as $p = (1, 2, \ldots, n)$ and whose entries are subsequently interchanged according to P_k. In addition, as was the case for Algorithm 2.2, the matrices `L` and `U` are actually superfluous, since their entries can be stored directly into `A`.

To leading order Algorithm 2.3 requires the same amount of floating point operations as Algorithm 2.2, that is, $2n^3/3$. In order to determine the pivot in step k, one has to look at $n-k$ entries. Thus, in total the partial pivoting strategy leads to additional cost of $\sum_{k=1}^{n-1}(n-k)$ operations, which is quadratic in n and therefore negligible compared to the already cubic cost. With complete pivoting, however, in every step $(n-k)^2$ entries must be examined, thus leading to an additional cost which is cubic in n and not negligible anymore.

Finally, concerning existence of a factorization $PA = LU$ we have the following result.

Theorem 2.6. *For every regular matrix $A \in \mathbb{K}^{n\times n}$ Gaussian elimination with partial pivoting produces a normalized LU factorization of PA, where $P \in \mathbb{K}^{n\times n}$ is a permutation matrix, and $|\ell_{ij}| \leq 1$.*

Stability

Stability analysis of Gaussian elimination is a complicated matter, which is why we will not go into details here. The gist is, however, that Gaussian elimination with partial pivoting is unstable in theory but perfectly stable in practice. Put differently, there are certain matrices for which it behaves in an unstable way, but these matrices are extremely rare in practice.

Theorem 2.7. *Let $A \in \mathbb{K}^{n\times n}$ be a regular matrix. Suppose A has normalized LU factorization $A = LU$, and that $\tilde{L}, \tilde{U}$ are the factors computed by Algorithm 2.2 on a machine satisfying (1.3) and (1.4). Then there is a matrix $\Delta A \in \mathbb{K}^{n\times n}$ such that*

$$\tilde{L}\tilde{U} = A + \Delta A \quad \text{and} \quad \|\Delta A\| \leq C\|L\|\|U\|u$$

for a moderate constant $C > 0$.

This theorem is to be interpreted in the following way. If $\|L\|\|U\| \approx \|A\|$, then $\|\Delta A\|/\|A\| \lesssim Cu$. In this case Gaussian elimination without pivoting is backward stable. Otherwise it is not. However, in Example 2.4 we have seen that $\|L\|$ and $\|U\|$ can become arbitrarily large. Therefore, Algorithm 2.2 is not backward stable.

For Gaussian elimination with partial pivoting there is a similar theorem.

Theorem 2.8. *Suppose the factorization $PA = LU$ of a regular matrix $A \in \mathbb{K}^{n\times n}$ is computed by Algorithm 2.3 on a machine satisfying (1.3) and (1.4). Denote the computed matrices by $\tilde{P}$, $\tilde{L}$ and $\tilde{U}$. Then there is a matrix $\Delta A \in \mathbb{K}^{n\times n}$ such that*

$$\tilde{L}\tilde{U} = \tilde{P}A + \Delta A \quad \text{and} \quad \|\Delta A\| \leq C\|U\|u$$

for a moderate constant $C > 0$.

Here, $\|L\|$ does not appear in the error estimate, since we know it is small (because of pivoting). Thus, Gaussian elimination with partial pivoting is backward stable if the ratio $\|U\|/\|A\|$ is moderate. For virtually all matrices A encountered in practice this is the case and therefore Algorithm 2.3 can be called backward stable in practice. Yet there are matrices where the ratio does become very large (see below). This is the reason why Gaussian elimination with partial pivoting is not backward stable in theory.

Example 2.6. Consider the following matrix

$$A = \begin{bmatrix} 1 & & & 1 \\ -1 & 1 & & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}.$$

Gaussian elimination gives

$$L = \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ -1 & -1 & 1 & \\ -1 & -1 & -1 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 1 & & & 1 \\ & 1 & & 2 \\ & & 1 & 4 \\ & & & 8 \end{bmatrix}.$$

Partial pivoting does not make a difference, since no row interchanges will occur for this particular A. Choosing the ∞-norm, for instance, the ratio mentioned in the previous paragraph equals

$$\frac{\|U\|_\infty}{\|A\|_\infty} = \frac{8}{4} = 2.$$

In general, however, for an $n \times n$ matrix with the same pattern as A the largest element of U will equal 2^{n-1}. Therefore the ratio equals $2^{n-1}/n$ which for growing n quickly becomes a huge number.

2.3 Cholesky Factorization

Self-adjoint positive definite matrices arise naturally in many physical systems. Recall from Example 2.2 that such matrices always have LU factorizations. In fact, they admit factorizations of the form $A = R^*R$, where R is upper triangular. Therefore only one matrix must be constructed. Accordingly, Cholesky factorizations can be computed twice as fast as general LU factorizations. The standard algorithm for doing so can be viewed as a symmetric variant of Gaussian elimination that operates on rows and columns at the same time.

2.3.1 SPD Matrices

Let us collect some properties of SPD matrices. The diagonal entries of SPD matrices are positive real numbers, because $a_{jj} = e_j^* A e_j > 0$ by positive definiteness. This fact is actually a special case of the following more general statement.

Lemma 2.3. *Let $A \in \mathbb{K}^{n\times n}$ be an SPD matrix and let $B \in \mathbb{K}^{n\times m}$, $n \geq m$, have full rank. Then B^*AB is an SPD matrix.*

Proof. The matrix B^*AB is self-adjoint, because

$$(B^*AB)^* = B^*A^*B^{**} = B^*AB.$$

Let $x \in \mathbb{K}^m$ be a nonzero vector. Then Bx is again nonzero, since B is of full rank. Therefore

$$x^*B^*ABx = (Bx)^*ABx > 0.$$

□

An important consequence of Lemma 2.3 is that every principal submatrix of an SPD matrix is again SPD. An $m \times m$ submatrix of A is called *principal submatrix*, if it is obtained from A by deleting any $n-m$ columns and the same $n-m$ rows.[3] Every principal submatrix of A can be written as a matrix product J^*AJ, where J is obtained from the n-dimensional identity matrix by deleting some columns. By Lemma 2.3 J^*AJ is SPD, if A is.

2.3.2 Symmetric Gaussian Elimination

Let $A \in \mathbb{K}^{n\times n}$ be an SPD matrix. Using block notation we can write it as

$$\begin{bmatrix} a & w^* \\ w & K \end{bmatrix},$$

where $a > 0$, $w \in \mathbb{K}^{n-1}$ and $K \in \mathbb{K}^{(n-1)\times(n-1)}$. One step of Gaussian elimination gives

$$L_1 A = \begin{bmatrix} 1 & 0 \\ -w/a & I \end{bmatrix} \begin{bmatrix} a & w^* \\ w & K \end{bmatrix} = \begin{bmatrix} a & w^* \\ 0 & K - ww^*/a \end{bmatrix}.$$

Next, instead of proceeding to the second column (as we would, were we computing an LU factorization), we apply a symmetric elimination step on the columns. That is, using column operations we eliminate the w^* in the first row of L_1A. Due to the self-adjointness of A, the matrix that performs this step is just L_1^*, which has to be multiplied from the right. Therefore, we get

$$L_1 A L_1^* = \begin{bmatrix} a & w^* \\ 0 & K - ww^*/a \end{bmatrix} \begin{bmatrix} 1 & -w^*/a \\ 0 & I \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & K - ww^*/a \end{bmatrix}.$$

Inverting the two matrices L_1 and L_1^* we have the following factorization of A

$$A = \underbrace{\begin{bmatrix} 1 & 0 \\ w/a & I \end{bmatrix}}_{L_1^{-1}} \underbrace{\begin{bmatrix} a & 0 \\ 0 & K - ww^*/a \end{bmatrix}}_{\tilde{A}_1} \underbrace{\begin{bmatrix} 1 & w^*/a \\ 0 & I \end{bmatrix}}_{L_1^{-*}}.$$

This is almost one step of Cholesky factorization. For reasons that will become clear soon, we would like the $(1,1)$ entry of $\tilde{A}_1$ to equal 1. Note that $\tilde{A}_1$ can be decomposed in the following way

$$\tilde{A}_1 = \begin{bmatrix} \sqrt{a} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a \end{bmatrix} \begin{bmatrix} \sqrt{a} & 0 \\ 0 & I \end{bmatrix}.$$

Recall that $a = a_{11} > 0$ and therefore $\sqrt{a}$ is again a real number, in particular a positive one. Plugging this product back into the previous factorization of A we finally arrive at

$$A = \underbrace{\begin{bmatrix} \sqrt{a} & 0 \\ w/\sqrt{a} & I \end{bmatrix}}_{R_1^*} \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & K - ww^*/a \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} \sqrt{a} & w^*/\sqrt{a} \\ 0 & I \end{bmatrix}}_{R_1}. \tag{2.7}$$

[3] In equation (2.5) we have already introduced a special type of principal submatrix.

If the submatrix $K - ww^*/a$ is SPD, then its $(1,1)$ entry must be a positive real number and we can eliminate the second row and column of A_1 by applying the above procedure to $K - ww^*/a$. This would give us a factorization $A_1 = R_2^* A_2 R_2$ and in combination with (2.7) we would get $A = R_1^* R_2^* A_2 R_2 R_1$. Continuing this process, after n steps we would eventually end up with

$$A = \underbrace{R_1^* \cdots R_n^*}_{R^*} \, I \, \underbrace{R_n \cdots R_1}_{R} = R^* R,$$

where $r_{jj} > 0$ for $j = 1, \ldots, n$. Such a factorization is called *Cholesky factorization*.

So, is $K - ww^*/a$ an SPD matrix? The answer is yes, because it is a principal submatrix of $A_1 = R_1^{-*} A R_1^{-1}$, which by Lemma 2.3 must be an SPD matrix. With the same reasoning it follows that for every k the (k,k) entry of A_{k-1} must be a positive number. The process we have described so far cannot fail and in fact produces a unique matrix R for every given SPD matrix A. Thus, we have the following theorem.

Theorem 2.9. *For every SPD matrix $A \in \mathbb{K}^{n\times n}$ there is an upper triangular matrix $R \in \mathbb{K}^{n\times n}$ such that $A = R^* R$. There is* exactly one *such matrix R satisfying $r_{jj} > 0$ for all j.*

The unique factorization into matrices with positive diagonal entries is usually referred to as the *Cholesky factorization* of A.

Example 2.7. Consider the SPD matrix

$$A = \begin{bmatrix} 4 & 2 & 0 \\ 2 & 3 & 3 \\ 0 & 3 & 9 \end{bmatrix}.$$

First, we subtract $\frac{1}{2}$ times the first row from the second and obtain

$$L_1 A = \begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 4 & 2 & 0 \\ 2 & 3 & 3 \\ 0 & 3 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 2 & 0 \\ 0 & 2 & 3 \\ 0 & 3 & 9 \end{bmatrix}.$$

Next we apply the same operations to the columns of $L_1 A$, that is, we subtract $\frac{1}{2}$ times the first column from the second

$$L_1 A L_1^* = \begin{bmatrix} 4 & 2 & 0 \\ 0 & 2 & 3 \\ 0 & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & -\frac{1}{2} & \\ & 1 & \\ & & 1 \end{bmatrix} = \begin{bmatrix} 4 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix}.$$

Now we invert the matrices L_1 and L_1^* and write A as a product of three matrices

$$A = \begin{bmatrix} 1 & & \\ \frac{1}{2} & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 4 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{2} & \\ & 1 & \\ & & 1 \end{bmatrix}$$
$$= \underbrace{\begin{bmatrix} 2 & & \\ 1 & 1 & \\ & & 1 \end{bmatrix}}_{R_1^*} \underbrace{\begin{bmatrix} 1 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} 2 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix}}_{R_1}.$$

We continue with the matrix A_1 and subtract $3/2$ times the second row from the third row, as well as $3/2$ times the second column from the third column. Using matrix notation this yields

$$L_2 A_1 L_2^* = \begin{bmatrix} 1 & & \\ & 1 & \\ & -\frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 2 & 3 \\ & 3 & 9 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & -\frac{3}{2} \\ & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix}.$$

But now A_1 can be expressed as

$$A_1 = \begin{bmatrix} 1 & & \\ & 1 & \\ & \frac{3}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 2 & \\ & & \frac{9}{2} \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \frac{3}{2} \\ & & 1 \end{bmatrix}$$
$$= \underbrace{\begin{bmatrix} 1 & & \\ & \sqrt{2} & \\ & \frac{3}{\sqrt{2}} & 1 \end{bmatrix}}_{R_2^*} \underbrace{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{9}{2} \end{bmatrix}}_{A_2} \underbrace{\begin{bmatrix} 1 & & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & 1 \end{bmatrix}}_{R_2}.$$

In order to finish the Cholesky factorization we only have to decompose

$$A_2 = \underbrace{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}}_{R_3^*} \underbrace{\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}}_{R_3}.$$

The factor R is now the product of the matrices R_1, R_2 and R_3

$$R = \begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{3}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 & \\ & \sqrt{2} & \frac{3}{\sqrt{2}} \\ & & \frac{3}{\sqrt{2}} \end{bmatrix}.$$

Note that there is no calculation needed in order to form R. Its k-th row is just the k-th row of R_k.

2.3.3 Algorithm

Let $A \in \mathbb{K}^{n \times n}$ be an SPD matrix and $k < n - 1$. After k steps of symmetric Gaussian elimination, we have reduced A to

$$A_k = \begin{bmatrix} I & \\ & S \end{bmatrix},$$

where I is a $k \times k$ identity matrix and S is an $(n-k) \times (n-k)$ SPD matrix which can be written as

$$S = \begin{bmatrix} s & w^* \\ w & \tilde{S} \end{bmatrix},$$

with $s > 0$. The $(k+1)$-th step now consists of two parts. One is computation of R_{k+1} and the other one is updating A_k into A_{k+1}. The general formulas are

$$A_{k+1} = \begin{bmatrix} I & \\ & \tilde{S} - ww^*/s \end{bmatrix},$$

$$R_{k+1} = \begin{bmatrix} I & & \\ & \sqrt{s} & w^*/\sqrt{s} \\ & & I \end{bmatrix}.$$

Note how the one nontrivial row of R_{k+1} is just the first row of S divided by $\sqrt{s}$. Finally, after $n-1$ steps A has been reduced to $A_k = A_{n-1} = \operatorname{diag}(1, \ldots, 1, s)$, where $s > 0$. No elimination is required anymore and the last matrix R_n is given by $R_n = \operatorname{diag}(1, \ldots, 1, \sqrt{s})$.

We can now write down the algorithm in Matlab syntax. It takes as input an $n \times n$ SPD matrix A and returns the upper triangular matrix R.

Algorithm 2.4: Cholesky Factorization

```
1 R = zeros(n);
2 for k = 1:n
3     for j = k+1:n
4         A(j,j:n) = A(j,j:n) - conj(A(k,j))/A(k,k) ...
5                     ... * A(k,j:n);
6     end
7     R(k,k:n) = A(k,k:n)/sqrt(A(k,k));
8 end
```

Let us compare the inner for-loop, which implements the elimination step, to that of Gaussian elimination without pivoting (Algorithm 2.2). First, the inner for-loop is empty, if `k = n`. Next, due to symmetry, the entries of `A` below the diagonal need not be updated. Finally, the ratio `conj(A(k,j))/A(k,k)` takes the role of `L(j,k)` in Alg. 2.2. (The command `conj` returns the complex conjugate of the argument.) As usual, explicitly forming the matrix `R` could be spared by writing its entries directly into the input array `A`.

Theorem 2.10. *To leading order Algorithm 2.4 requires $n^3/3$ operations.*

Proof. The proof is very similar to previous ones about operation counting. We therefore omit it. □

When computing Cholesky-factorizations there is no need for pivoting, because Algorithm 2.4 is always stable. The problems encountered for the LU factorization, where the factors U might have very large norms compared to A, cannot occur here. In fact, one can show that

$$\|R\|_2 = \|R^*\|_2 = \|A\|_2^{\frac{1}{2}}.$$

Theorem 2.11. *Algorithm 2.4 is backward stable. More precisely, for every SPD matrix $A \in \mathbb{K}^{n\times n}$ Algorithm 2.4 implemented on a machine satisfying (1.3) and (1.4) produces an upper triangular matrix $\tilde{R}$ such that*

$$\tilde{R}^*\tilde{R} = A + \Delta A,$$

where the error matrix $\Delta A \in \mathbb{K}^{n\times n}$ satisfies

$$\frac{\|\Delta A\|}{\|A\|} \leq Cu,$$

for some moderate $C > 0$.

2.4 QR Factorization

In this section we do not restrict our attention to square matrices, but instead consider $A \in \mathbb{K}^{m\times n}$, $m \geq n$. For such matrices there are two different types of QR factorizations, a reduced and a full one. The *reduced QR factorization*

$$A = \hat{Q}\hat{R}$$

expresses A as a product of an $m \times n$ matrix $\hat{Q}$ with orthonormal columns and an upper triangular matrix $\hat{R} \in \mathbb{K}^{n\times n}$. For the *full QR factorization*

$$A = QR$$

the matrix $Q \in \mathbb{K}^{m\times m}$ is orthogonal/unitary and $R \in \mathbb{K}^{m\times n}$ is generalized upper triangular, that is, it is not necessarily square but still $r_{ij} = 0$ for $i > j$. The relation between these two factorization can be visualized as follows

$$A = \underbrace{\begin{bmatrix}\hat{Q} & \cdots\end{bmatrix}}_{Q} \underbrace{\begin{bmatrix}\hat{R}\\ 0\end{bmatrix}}_{R}.$$

The matrix Q contains an additional $m - n$ orthonormal vectors so that its columns form an orthonormal basis of $\mathbb{K}^m$. However, these additional columns do not contribute to the product QR since they are multiplied with the block of zeros at the bottom of R. Clearly, if $m = n$, then there is no difference between reduced and full QR factorization.

2.4.1 Gram-Schmidt

The QR factorization can be understood in terms of a well-known algorithm from linear algebra: the Gram-Schmidt process. See Section A.4.

First, let us write out a reduced QR factorization using column vectors

$$\underbrace{\left[\begin{array}{c|c|c} & & \\ a_1 & \cdots & a_n \\ & & \end{array}\right]}_{A} = \underbrace{\left[\begin{array}{c|c|c} & & \\ q_1 & \cdots & q_n \\ & & \end{array}\right]}_{\hat{Q}} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}}_{\hat{R}}.$$

Writing out separate equations for every column of A yields the following system

$$\begin{aligned} a_1 &= r_{11} q_1 \\ a_2 &= r_{12} q_1 + r_{22} q_2 \\ &\vdots \\ a_n &= r_{1n} q_1 + r_{2n} q_2 + \cdots + r_{nn} q_n. \end{aligned} \tag{2.8}$$

Since every column of A is expressed as a linear combination of certain columns of $\hat{Q}$, we have that

$$a_k \in \operatorname{span}\{q_1, \ldots, q_k\}, \quad k = 1, \ldots, n.$$

Assume for the moment that A has full rank. This implies that also $\hat{R}$ has full rank and therefore all diagonal entries r_{kk} are nonzero. In particular the $k \times k$ leading principal submatrix of $\hat{R}$ is invertible so that we can express q_k as linear combinations of the first k columns of A. Thus,

$$q_k \in \operatorname{span}\{a_1, \ldots, a_k\}, \quad k = 1, \ldots, n,$$

and therefore the columns of A and $\hat{Q}$ span the same spaces

$$\operatorname{span}\{a_1, \ldots, a_k\} = \operatorname{span}\{q_1, \ldots, q_k\}, \quad k = 1, \ldots, n. \tag{2.9}$$

We can now reformulate the problem of finding a reduced QR factorization for a given full rank matrix $A \in \mathbb{K}^{m \times n}$: Given n linearly independent vectors $a_1, \ldots, a_n \in \mathbb{K}^m$, find n orthonormal vectors $q_1, \ldots, q_n \in \mathbb{K}^m$ satisfying (2.9). But this is exactly what the Gram-Schmidt process does! Thus, we already know an algorithm to compute the q_k.

What about the r_{ij}? Let us first express the q_k in (2.8)

$$\begin{aligned} q_1 &= \frac{a_1}{r_{11}} \\ q_2 &= \frac{1}{r_{22}} (a_2 - r_{12} q_1) \\ &\vdots \\ q_n &= \frac{1}{r_{nn}} (a_n - r_{1n} q_1 - \cdots - r_{n-1,n} q_{n-1}). \end{aligned}$$

Now we only have to compare with the Gram-Schmidt process, see eq. (A.2), to find that

$$r_{ij} = \begin{cases} q_i^* a_j, & i < j \\ \| a_j - r_{1j} q_1 - \cdots - r_{j-1,j} q_{j-1} \|_2, & i = j. \end{cases}$$

Having found general formulas for both q_k and r_{ij} we can write down an algorithm which for a given full rank matrix $A \in \mathbb{K}^{m \times n}$ computes a reduced QR factorization.

Algorithm 2.5: Classical Gram-Schmidt

```
R = zeros(n);
Q = zeros(m,n);
V = A;
for j=1:n
    for i = 1:j-1
            R(i,j) = Q(:,i)' * A(:,j);
            V(:,j) = V(:,j) - R(i,j) * Q(:,i);
    end
    R(j,j) = norm(V(:,j));
    Q(:,j) = V(:,j)/R(j,j);
end
```

Theorem 2.12. *Every $A \in \mathbb{K}^{m\times n}$, $m \geq n$, has both a reduced and a full QR factorization.*

Proof. If A has full rank, then Algorithm 2.5 cannot break down, because $r_{jj} \neq 0$, thus producing a reduced QR factorization. Extend $\hat{Q}$ to an orthonormal basis and append $m-n$ zero rows to $\hat{R}$ to obtain a full QR factorization.

If A does not have full rank, then $r_{jj} = 0$ for some j. In this case set q_j equal to any normalized vector orthogonal to all previously computed q_i and proceed to $j+1$. □

The second part of this proof implies that a QR factorization cannot be unique, if A is rank-deficient. Moreover, regardless of the rank of A full QR factorizations can actually never be unique, since there are many ways to extend n orthonormal vectors to an orthonormal basis (unless $m = n$, of course).

Theorem 2.13. *Every full rank matrix $A \in \mathbb{K}^{m\times n}$, $m \geq n$, has a unique reduced QR factorization with $r_{jj} > 0$ for all $j = 1, \ldots, n$.*

Proof. Since A has full rank, the algorithm does not fail. However, we could set $r_{jj} = -\|\cdots\|$ and obtain a different set of orthonormal vectors. Upon fixing these signs, the factorization becomes unique. □

Unfortunately, the Gram-Schmidt process is well-known for its numerical instability. In particular it suffers from a significant *numerical loss of orthogonality,* especially when A is close to rank-deficient. The following example[4] illustrates this behaviour.

Example 2.8. Consider the matrix

$$A = \begin{bmatrix} 0.70000 & 0.70711 \\ 0.70001 & 0.70711 \end{bmatrix}.$$

[4] This example is taken from [16]

On a machine with a precision of 5 digits r_{11} is computed to be $r_{11} = 0.98995$. The first column of Q equals

$$q_1 = a_1/r_{11} = \begin{bmatrix} 0.70711 \\ 0.70712 \end{bmatrix}.$$

Next, $r_{12} = 1.0000$ and finally, by means of cancellation, the rounding errors become prevalent

$$a_2 - r_{12}q_1 = \begin{bmatrix} 0.00000 \\ -0.00001 \end{bmatrix}, \quad q_2 = \begin{bmatrix} 0.0000 \\ -1.0000 \end{bmatrix}.$$

Clearly, q_1 and q_2 are far from orthogonal.

2.4.2 Householder's Method

The basic idea of Householder's method for finding QR factorizations is to transform A into generalized upper triangular form R via multiplication from the left with orthogonal/unitary matrices

$$\underbrace{Q_n \cdots Q_1}_{Q^*} A = R.$$

As a product of orthogonal/unitary matrices Q^* is itself orthogonal/unitary and so is its inverse

$$Q = Q^{**} = Q_1^* \cdots Q_n^*,$$

and therefore $A = QR$ is a full QR factorization.

The matrix Q_k should be such that it introduces zeros in the k-th column below the main diagonal while leaving previously introduced zeros untouched, for example

$$\underset{A}{\begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1A}{\begin{bmatrix} * & * & * \\ & * & * \\ & * & * \\ & * & * \end{bmatrix}} \xrightarrow{Q_2} \underset{Q_2Q_1A}{\begin{bmatrix} * & * & * \\ & * & * \\ & & * \\ & & * \end{bmatrix}} \xrightarrow{Q_3} \underset{Q_3Q_2Q_1A=R}{\begin{bmatrix} * & * & * \\ & * & * \\ & & * \\ & & \end{bmatrix}}.$$

If Q_k is to leave the first $k-1$ columns unchanged while being an orthogonal/unitary matrix, it should have the following block structure

$$Q_k = \begin{bmatrix} I & \\ & H_k \end{bmatrix}, \tag{2.10}$$

where I is the $(k-1)$-dimensional identity matrix and H_k is another $(m-k+1) \times (m-k+1)$ orthogonal/unitary matrix. On the other hand, the matrix we are operating on in the k-th step should look like this

$$Q_{k-1} \cdots Q_1 A = \begin{bmatrix} U & B \\ & C \end{bmatrix},$$

where U is a $(k-1)\times(k-1)$ upper triangular matrix and B, C have no particular structure. Therefore the result of step k reads as follows

$$\underbrace{\begin{bmatrix} I & \\ & H_k \end{bmatrix}}_{Q_k} \underbrace{\begin{bmatrix} U & B \\ & C \end{bmatrix}}_{Q_{k-1}\cdots Q_1 A} = \underbrace{\begin{bmatrix} U & B \\ & H_k C \end{bmatrix}}_{Q_k \cdots Q_1 A}.$$

The matrix H_k should introduce zeros in the first column c_1 of C, more precisely it should turn the first column of C into a multiple of $e_1 = (1, 0, \ldots, 0)^\top$

$$H_k c_1 = s e_1,$$

where $s \in \mathbb{K}$. However, H_k is orthogonal/unitary and therefore preserves the 2-norm, recall equation (A.3) in the appendix. This implies that

$$\|c_1\|_2 = \|H_k c_1\|_2 = \|s e_1\|_2 = |s|.$$

Therefore, the scalar s is not arbitrary, but must equal $\sigma\|c_1\|_2$, where $|\sigma| = 1$. If $\mathbb{K} = \mathbb{R}$, then there are two possibilities for s. If $\mathbb{K} = \mathbb{C}$, then there are infinitely many.

Supposing that we have fixed a value for σ, there is more than one orthogonal/unitary transformation H_k that maps the vector c_1 to $\sigma\|c_1\|_2 e_1$. The Householder method selects H_k as the matrix that reflects across the hyperplane orthogonal to the vector

$$v = c_1 - \sigma\|c_1\|_2 e_1. \tag{2.11}$$

This so-called *Householder reflection* is given by

$$H_k = I - 2\frac{vv^*}{v^*v}. \tag{2.12}$$

See Fig. 2.1 for a two-dimensional sketch.

The final question we need to answer is how to choose the σ in (2.11). The guiding principle for this decision is numerical stability. More specifically, in order to avoid cancellation in (2.11) we should choose σ such that v is as long as possible. Therefore

$$\sigma = -\frac{c_{11}}{|c_{11}|}. \tag{2.13}$$

If $c_{11} = 0$, we can choose σ arbitrarily, e.g. equal to one. In this case cancellation cannot occur.

Formulas (2.10), (2.11), (2.12) and (2.13) completely define the orthogonal transformation Q_k which we want to apply in step k of the Householder method. Thus we are in a position to write down the algorithm. The following Matlab code takes as input an $m \times n$ matrix A where $m \geq n$.

Algorithm 2.6: Householder Method

```
1 V = zeros(m,n);
2 for k = 1:n
```

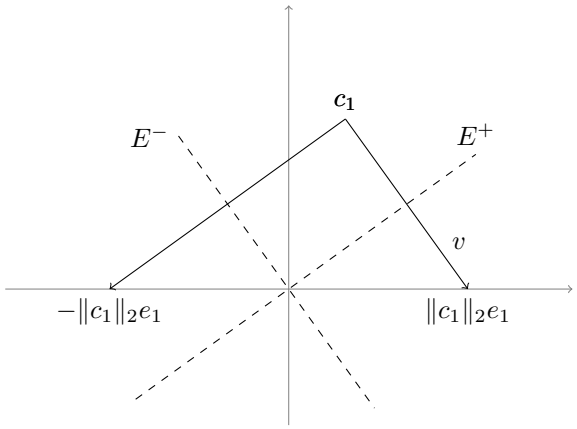

Figure 2.1: 2D sketch of the Householder reflection. E^+ is the hyperplane (i.e. line) orthogonal to $v = c_1 - \|c_1\|_2 e_1$, whereas E^- is the hyperplane orthogonal to $c_1 + \|c_1\|_2 e_1$.

```
3       v =  A(k:m,k);
4       v(1) = v(1) + sign(v(1))*norm(v);
5       v = v/norm(v);
6       A(k:m,k:n) = A(k:m,k:n) - 2*v*(v'*A(k:m,k:n));
7       V(1:m-k+1,k) = v;
8   end
9   R=A;
```

Remark 2.4. For the sake of brevity the algorithm, as it is stated, neglects two issues. First, an exception is required to take care of the possibility that $c_{11} = 0$ (see above). Second, it might happen that *all* entries of c_1 equal zero, which would lead to a division by zero in line 5. However, $c_1 = 0$ means that the column under consideration already has the desired form and can be left as it is. Therefore, more generally, one should proceed as follows: If c_1 already is a multiple of e_1, skip the current iteration of the for-loop and continue with $k+1$.

Note that Algorithm 2.6 does not construct the matrices Q_k or Q explicitly, but only the vectors v. It takes additional work to construct Q, but often this is not necessary. First, recall that

$$Q^* = Q_n \cdots Q_1 \qquad \text{and} \qquad Q = Q_1^* \cdots Q_n^* = Q_1 \cdots Q_n, \tag{2.14}$$

the last equality being due to the fact that the Householder reflections H_k are self-adjoint and therefore the Q_k are as well. Now, if we needed to compute the matrix-vector product Q^*b for some $b \in \mathbb{K}^m$ we could do so without forming Q^* with the following lines of code.

Algorithm 2.7: Calculation of Q^*b

```
for k = 1:n
    b(k:m) = b(k:m) - ...
         ... 2*V(1:m-k+1,k)*(V(1:m-k+1,k)'*b(k:m));
end
```

Or, if we wanted to compute Qx for some $x \in \mathbb{K}^m$:

Algorithm 2.8: Calculation of Qx

```
for k = n:-1:1
    x(k:m) = x(k:m) - ...
         ... 2*V(1:m-k+1,k)*(V(1:m-k+1,k)'*x(k:m));
end
```

Note how, because of (2.14), the index `k` increases in the first snippet of code but decreases in the second one.

Theorem 2.14. *To leading order Algorithm 2.6 requires* $2mn^2 - \frac{2}{3}n^3$ *operations.*

Theorem 2.15. *The Householder method is backward stable. More precisely, for every* $A \in \mathbb{K}^{m\times n}$ *with* $m \geq n$*, Algorithm 2.6 implemented on a machine satisfying* (1.3) *and* (1.4) *computes factors* $\tilde{Q}$ *and* $\tilde{R}$ *satisfying*

$$\tilde{Q}\tilde{R} = A + \Delta A,$$

where the error matrix ΔA *is such that*

$$\frac{\|\Delta A\|}{\|A\|} \leq Cu$$

for some moderate $C > 0$*.*

As stated the theorem above is sloppy. Recall that Algorithm 2.6 does *not* compute $\tilde{Q}$. Instead it explicitly avoids computation of the orthogonal matrix as this would mean additional work. How should the theorem be interpreted then? The algorithm *does* compute floating point approximations $\tilde{v}$ of the vectors v defined by (2.11) and (2.13). Therefore, the $\tilde{Q}$ in Theorem 2.15 is the exactly unitary matrix satisfying $\tilde{Q} = \tilde{Q}_1 \cdots \tilde{Q}_n$, where the $\tilde{Q}_k$ are the matrices defined *mathematically* by equations (2.10) and (2.12), given $\tilde{v}$.

2.4.3 Comparison of Direct Methods

Suppose A is a regular matrix. In Chapter 2 we have so far encountered three matrix factorizations. Combined with forward and/or back substitution each of them gives rise to a different method for solving a system of linear equations $Ax = b$.

LU factorization

1. Factorize $PA = LU$

2. Solve $Ly = Pb$ for y via forward substitution
3. Solve $Ux = y$ for x via back substitution

Cholesky factorization

1. Factorize $A = R^*R$
2. Solve $R^*y = b$ for y via forward substitution
3. Solve $Rx = y$ for x via back substitution

QR factorization

1. Factorize $A = QR$
2. Compute the vector $y = Q^*b$
3. Solve $Rx = y$ for x via back substitution

Cholesky factorization can only be used if A is an SPD matrix. In this case, however, it is the method of choice. It is always backward stable and is the fastest of three methods, only requiring $n^3/3$ operations. If A is not SPD, then LU factorization is the standard method. It is backward stable in practice and requires $2n^3/3$ operations.

If A is ill-conditioned, then QR factorization is a good choice. In Example 1.10 we have shown that the condition number κ_2 is invariant under orthogonal/unitary transformations. Therefore $\kappa_2(A) = \kappa_2(QR) = \kappa_2(R)$. This means that the QR approach does not lead to additional error amplification. (Recall our brief discussion about condition numbers at the beginning of Chapter 2 on page 17.) The Householder method is a backward stable algorithm for computing QR factorizations. For square matrices it requires $4n^3/3$ operations and is therefore twice as costly as LU factorization. In the next section we will encounter another application for the QR factorization.

Remark 2.5. In Matlab and Octave linear systems of equations can be solved with the command `A\b` or, equivalently, `mldivide(A,b)`. When entering these commands Matlab/Octave decides which algorithm to actually apply. It is insightful to understand how this decision making is done. See, for instance, Matlab's online documentation on this topic.[5]

2.5 Linear Least Squares Problems

Linear least squares problems are a fundamental class of problems with a wide range of applications throughout the mathematical sciences. In finite dimensions it can be stated as follows. Let $m \geq n$. For given matrix $A \in \mathbb{K}^{m\times n}$ and right hand side $b \in \mathbb{K}^m$ find a vector $x \in \mathbb{K}^n$ that minimizes $\|Ax - b\|_2$. Note that minimizing $\|Ax - b\|_2$ is equivalent to minimizing $\|Ax - b\|_2^2$. Hence the name *least squares*.

[5] https://www.mathworks.com/help/matlab/ref/mldivide.html

For $m > n$ the overdetermined system $Ax = b$ does not have a solution in general. It only does, if $b \in \operatorname{ran} A$, which is an n-dimensional subspace of $\mathbb{K}^m$. Therefore, minimizing the norm of the *residual* $Ax - b$ is, in some sense, the best one can do. Of course, the choice of the 2-norm is arbitrary and different norms will lead to different solutions. The 2-norm gives rise to a solution $x \in \mathbb{K}^n$ such that Ax is the closest point (in the Euclidean sense) in $\operatorname{ran} A$ to b.

The following theorem gives two equivalent characterizations of solutions to the linear least squares problem as stated above.

Theorem 2.16. *Let $m \geq n$, $A \in \mathbb{K}^{m\times n}$ and $b \in \mathbb{K}^m$. A vector $x \in \mathbb{K}^n$ solves the linear least squares problem, that is, it satisfies*

$$\|Ax - b\|_2 \leq \|Ay - b\|_2, \quad \text{for all } y \in \mathbb{K}^n, \tag{2.15}$$

if and only if it solves the normal equations

$$A^*Ax = A^*b, \tag{2.16}$$

or equivalently

$$Ax = Pb, \tag{2.17}$$

where P is the orthogonal projection onto the range of A. The solution x is unique, if and only if A has full rank.

Proof. This proof is structured as follows. In the first paragraph below we show that x is a solution, if and only if (2.17) is satisfied. The second paragraph is devoted to proving equivalence of (2.16) and (2.17). Finally, we address the uniqueness of x.

First of all we need a generalization of Pythagoras' theorem: Let $x, y \in \mathbb{K}^m$ be a pair of orthogonal vectors, that is, $x^*y = 0$. Then

$$\|x + y\|_2^2 = (x + y)^*(x + y) = x^*x + x^*y + y^*x + y^*y = \|x\|_2^2 + \|y\|_2^2.$$

That is, the sum of squared norms equals the squared norm of the sum. Next, we use this identity to rewrite the squared norm of the residual

$$\|Ax - b\|_2^2 = \|Ax - Pb + Pb - b\|_2^2 = \|Ax - Pb\|_2^2 + \|Pb - b\|_2^2. \tag{2.18}$$

Why does Pythagoras' theorem apply here? We need to verify that the two vectors $Ax - Pb$ and $Pb - b$ are orthogonal. The first vector $Ax - Pb \in \operatorname{ran} A = \operatorname{ran} P$, since P by definition maps onto the range of A. For the second vector we have $Pb - b = (P - I)b \in \operatorname{ran}(I - P)$. Recall (or see Sect. A.5) that the range of an orthogonal projection is always orthogonal to the range of its complementary projection $I - P$. Therefore, splitting the norm in (2.18) is justified. But this means that minimization of $\|Ax - b\|_2^2$ is equivalent to minimization of $\|Ax - Pb\|_2^2$, because $\|Pb - b\|_2^2$ is independent of x. Finally, minimizing $\|Ax - Pb\|_2^2$ is equivalent to simply solving $Ax = Pb$, which always has a solution, since $Pb \in \operatorname{ran} A$. Thus we have shown that (2.15) and (2.17) are equivalent.

Another way of writing the equation $Ax = Pb$ is $P(Ax - b) = 0$. This means that the residual vector $Ax - b$ lies in the kernel of P. However, since P is the orthogonal projection onto the range of A we have

$$\ker P \perp \operatorname{ran} P = \operatorname{ran} A.$$

Thus, the residual $Ax - b$ is orthogonal to the range of A. Since the range of A is nothing but the column space of A, we can express this orthogonality as

$$a_j^*(Ax - b) = 0, \quad \text{for } j = 1, \dots, n,$$

where a_j is the j-th column of A. Yet another way of writing this is

$$A^*(Ax - b) = 0,$$

which is just the system of normal equations (2.16).

Finally, x is the unique solution of the least squares problem, if and only if (2.16) has a unique solution. But this is equivalent to the matrix A^*A being regular, which is again equivalent to A having full rank. □

2.5.1 Numerical Solution

There are several ways to solve the linear least squares problem numerically. In this section we consider two of them. For both we assume that A has full rank.

One possibility is to exploit the fact that A^*A is an SPD matrix, if A has full rank, and to solve the normal equations using Cholesky factorization. The main steps for this approach are:

1. Compute A^*A and A^*b
2. Compute the Cholesky factorization $A^*A = R^*R$
3. Solve $R^*y = A^*b$ via forward substitution
4. Solve $Rx = y$ via back substitution

The work for this approach is dominated by the computation of the matrix A^*A, which requires mn^2 operations, and the Cholesky factorization. To leading order the total work required is therefore $mn^2 + n^3/3$. Solving the normal equations is fast, but the computation of the matrix A^*A can lead to numerical instabilities.

Another possibility is to use QR factorization. Here, we can exploit the fact that, if A has reduced QR factorization $A = \hat{Q}\hat{R}$, then the orthogonal projection P is given by $P = \hat{Q}\hat{Q}^*$. In this case the system (2.17) simplifies to

$$Ax = Pb \quad \Leftrightarrow \quad \hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b \quad \Leftrightarrow \quad \hat{R}x = \hat{Q}^*b.$$

The main algorithmic steps are therefore:

1. Compute a reduced QR factorization $A = \hat{Q}\hat{R}$

2. Compute the vector $\hat{Q}^*b$
3. Solve $\hat{R}x = \hat{Q}^*b$ via back substitution

The work for this approach is dominated by the QR factorization. If the Householder method is used, it requires $\sim 2mn^2 - 2n^3/3$ operations, which is more than the $\sim mn^2 + n^3/3$ operations required by the Cholesky-based approach. It does, however, avoid computation of A^*A and is therefore more stable in general. Regarding step 2, note that the vector $\hat{Q}^*b$ consists of the first n entries of Q^*b and can therefore efficiently computed with Algorithm 2.7.

Finally, the least squares problem can also be solved using the singular value decomposition of A, which will be introduced in a later chapter. When A is almost rank-deficient, this approach is typically superior in terms of stability to the QR-based approach.

Chapter 3

Interpolation

One way to think of interpolation is as a special case of data fitting. Suppose you are given n data points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{K}^2$. *Data fitting* is the problem of finding a (simple) function $\phi : \mathbb{K} \to \mathbb{K}$ that captures the trend of the data. This could be achieved by minimizing a cost function such as $E[\phi] = \sum_i |\phi(x_i) - y_i|^2$ over a set Φ of admissible functions. The choice of cost function and which functions are admissible depends a lot on the considered application. For example, if the data are oscillatory, then trigonometric functions ϕ might be a good choice. On the other hand, if the data display a decay behaviour, then functions of the form $\phi(x) = ae^{-\lambda x} + be^{-\mu x}$ might be appropriate.

Data interpolation is a special type of data fitting, where you require the function ϕ to exactly pass through all the data points, that is,

$$\phi(x_i) = y_i \quad \text{for } i = 1, \ldots, n.$$

In this chapter we will consider interpolation with three types of functions: polynomials, splines, and trigonometric polynomials.

3.1 Polynomial Interpolation

We start with a simple example.

Example 3.1 (Interpolating four points with a cubic polynomial.)**.** Consider the data points $(-2, 10)$, $(-1, 4)$, $(1, 6)$ and $(2, 3)$. We want to find a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3.1}$$

satisfying the four equations

$$p(-2) = 10, \quad p(-1) = 4, \quad p(1) = 6, \quad p(2) = 3. \tag{3.2}$$

The first question we need to address concerns the *degree* of p. A polynomial of degree n or less, like the one in (3.1), is entirely determined by its $n + 1$ coefficients $a_0, \ldots, a_n$. A reasonable guess is that we should look for a polynomial of degree $n = 3$, since in that case we have as many unknowns as equations.

Each of the four interpolation conditions in (3.2) is *linear* in the coefficients a_0, a_1, a_2, a_3. Therefore (3.2) can be written as

$$\begin{bmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 6 \\ 3 \end{bmatrix}.$$

It turns out that this matrix is regular and we have a unique solution

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{12} \begin{bmatrix} 54 \\ 23 \\ 6 \\ -11 \end{bmatrix}.$$

So the interpolating polynomial is given by

$$p(x) = \frac{9}{2} + \frac{23}{12}x + \frac{1}{2}x^2 - \frac{11}{12}x^3.$$

3.1.1 The General Problem

In the rest of this section we assume to be given $n+1$ real data points

$$(x_0, y_0), \ldots, (x_n, y_n) \in \mathbb{R}^2.$$

In addition we assume that where all x_i are distinct. Motivated by Exercise 3.1 we look for a polynomial p of degree n such that

$$p(x_i) = y_i \quad \text{for } i = 0, \ldots, n.$$

Plugging the $n+1$ data points into the general formula for $p(x)$ leads to the following system of $n+1$ linear equations

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{3.3}$$

The $(n+1)\times(n+1)$ matrix arising in this way is called *Vandermonde matrix.* The polynomial interpolation problem has a unique solution, if the Vandermonde matrix is regular.

Theorem 3.1. *A Vandermonde matrix is regular if and only if the x_i are distinct.*

Proof. A square matrix A is regular, if the zero vector $x = 0$ is the only vector satisfying $Ax = 0$. (Recall Theorem A.1.)

Denote the matrix in (3.3) by V. Suppose $a \in \mathbb{R}^{n+1}$ is a coefficient vector satisfying $Va = 0$. An equivalent way of writing this is

$$\sum_{i=0}^{n} a_i x_j^i = 0 \quad \text{for } j = 0, \ldots, n.$$

But this means that the polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

which is of degree $\leq n$, has $n+1$ zeros. (Recall that all x_j are distinct.) This contradicts the Fundamental Theorem of Algebra, which states that p cannot have more than n zeros, unless p is the zero polynomial. Thus p must be the zero polynomial, or equivalently $a = 0$. □

In summary, the polynomial interpolation problem reduces to a system of linear equations, which always has a unique solution, provided we look for a polynomial of the right degree. Therefore, at least from a theoretical perspective, we have solved the polynomial interpolation problem. In practice, however, it turns out to be more intricate, since Vandermonde matrices tend to have large condition numbers.

Above, in equation (3.1), we have chosen to represent the polynomial p as a linear combination of the "canonical" basis $\{1, x, x^2, \ldots, x^n\}$. Yet there are many other bases in which polynomials can be expressed and the choice of basis strongly influences the condition number of the resulting Vandermonde matrix.

Lemma 3.1. *Denote by $\mathbb{P}_n$ the set of all polynomials $p : \mathbb{R} \to \mathbb{R}$ of degree $\leq n$.*

1. *The set $\mathbb{P}_n$ is a real vector space.*
2. *The dimension of $\mathbb{P}_n$ is $n+1$.*
3. *The* monomials $1, x, x^2, \ldots, x^n$ *form a basis of $\mathbb{P}_n$.*
4. *The* Newton polynomials

$$\omega_j(x) = \begin{cases} 1, & j = 0, \\ \prod_{k=0}^{j-1} (x - x_k), & j = 1, \ldots, n. \end{cases}$$

   form a basis of $\mathbb{P}_n$.

5. *The* Lagrange polynomials

$$L_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}, \quad j = 0, \ldots, n.$$

   form a basis of $\mathbb{P}_n$.

Rigorously speaking, item 1 above says that the set $\mathbb{P}_n$ together with the two operations of (i) addition (of polynomials) and (ii) multiplication (of polynomials with real numbers) satisfies the vector space axioms.[1] Intuitively speaking, it simply says that every linear combination of polynomials of degree n or less is another polynomial of degree n or less. Consequently, notions like linear (in)dependence, basis and dimension make sense for polynomials, see Section A.1. The statement

"The polynomials $q_1, \ldots, q_m \in \mathbb{P}_n$ are linearly independent."

means that the only linear combination $\sum_{i=1}^{m} a_i q_i$ that returns the zero polynomial is the one where all $a_i = 0$. Similarly,

"The polynomials $q_0, \ldots, q_n \in \mathbb{P}_n$ form a basis of $\mathbb{P}_n$."

means that they are linearly independent and that we can write every polynomial of order $\leq n$ as a linear combination $\sum_{i=0}^{n} a_i q_i$. Finally,

"$\mathbb{P}_n$ has dimension $n+1$."

means that every basis of $\mathbb{P}_n$ must consist of $n+1$ polynomials.

Note that, in contrast to the monomial basis, the Newton and Lagrange polynomials depend on the x_i.

Example 3.2. Consider the following interpolation points

$$\begin{aligned}(x_0, y_0) &= (1, 1),\\ (x_1, y_1) &= (2, 4),\\ (x_2, y_2) &= (3, 9).\end{aligned}$$

The associated Newton polynomials

$$\begin{aligned}\omega_0(x) &= 1,\\ \omega_1(x) &= x - x_0 = x - 1,\\ \omega_2(x) &= (x - x_0)(x - x_1) = (x - 1)(x - 2)\end{aligned}$$

form a basis of $\mathbb{P}_2$. Therefore, the unique quadratic interpolating polynomial can be written as

$$\begin{aligned}p(x) &= a_0\omega_0(x) + a_1\omega_1(x) + a_2\omega_2(x)\\ &= a_0 + a_1(x - 1) + a_2(x - 1)(x - 2).\end{aligned}$$

Plugging in the three interpolation points leads to the following system

$$\begin{bmatrix}1 & & \\ 1 & 1 & \\ 1 & 2 & 2\end{bmatrix}\begin{bmatrix}a_0\\ a_1\\ a_2\end{bmatrix} = \begin{bmatrix}1\\ 4\\ 9\end{bmatrix},$$

[1] See, for instance, https://en.wikipedia.org/wiki/Vector_space.

which has solution $a = (1, 3, 1)^\top$ and therefore $p(x) = x^2$.

On the other hand, the Lagrange polynomials associated to the interpolation points (x_i, y_i) are given by

$$\begin{aligned} L_0(x) &= \frac{x - x_1}{x_0 - x_1}\frac{x - x_2}{x_0 - x_2} = \frac{x-2}{1-2}\frac{x-3}{1-3} = \frac{1}{2}(x-2)(x-3), \\ L_1(x) &= \frac{x - x_0}{x_1 - x_0}\frac{x - x_2}{x_1 - x_2} = \frac{x-1}{2-1}\frac{x-3}{2-3} = -(x-1)(x-3), \\ L_2(x) &= \frac{x - x_0}{x_2 - x_0}\frac{x - x_1}{x_2 - x_1} = \frac{x-1}{3-1}\frac{x-2}{3-2} = \frac{1}{2}(x-1)(x-2). \end{aligned}$$

They also form a basis of $\mathbb{P}_2$, which means we can write

$$p(x) = b_0 L_0(x) + b_1 L_1(x) + b_2 L_2(x).$$

This time we obtain the system

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 9 \end{bmatrix}.$$

The system is very easy to solve this time. We have $b = y$ and therefore

$$p(x) = \sum_{i=0}^{2} y_i L_i(x) = x^2.$$

Remark 3.1. In the previous example the Vandermonde matrix for the Newton basis turned out to be lower triangular. In fact, this is always the case, since the Newton polynomials have the property

$$\omega_j(x_i) = 0 \quad \text{for } i < j.$$

Similarly, the Lagrange polynomials satisfy

$$L_j(x_i) = \delta_{ij},$$

and therefore they always let you write down the interpolating polynomial directly in terms of the y_i without any additional calculations required

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \tag{3.4}$$

We summarize the solution of the polynomial interpolation problem in an arbitrary basis with the following theorem.

Theorem 3.2. *Let $x_0, \ldots, x_n \in \mathbb{R}$ be pairwise distinct and $y_0, \ldots, y_n \in \mathbb{R}$ arbitrary. Suppose that $\{q_0, \ldots, q_n\}$ is a basis of $\mathbb{P}_n$. Then the unique interpolating polynomial $p \in \mathbb{P}_n$ is given by*

$$p(x) = \sum_{i=0}^{n} a_i q_i(x),$$

where the vector of coefficients $(a_0, \dots, a_n)^\top$ *is the unique solution of*

$$\begin{bmatrix} q_0(x_0) & \cdots & q_n(x_0) \\ \vdots & & \vdots \\ q_0(x_n) & \cdots & q_n(x_n) \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}.$$

Proof. The only thing to prove is that the system matrix is invertible for every basis, which can be done as in the proof of Theorem 3.1. □

Which basis is a good practical choice? The Lagrange interpolant in its most obvious form,

$$p(x) = \sum_{j=0}^{n} y_j \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}, \tag{3.5}$$

while being useful theoretically, has several drawbacks. First, it is computationally expensive. The number of flops required to evaluate it at a single point x is proportional to n^2. Second, if a new data point (x_{n+1}, y_{n+1}) is added, all computations have to be redone. Third, evaluation of (3.5) is unstable.

One option is to use the *barycentric* form of the Lagrange interpolant, which is more efficient and stable, see [3]. Another possibility is the method of *divided differences* for the Newton basis [14]. Finally, we mention that the Matlab function `polyfit` solves the system (3.3) arising from the monomial basis.[2] See also Remark 3.2.

3.1.2 Error Estimates

Assuming that the y_i are the values of some function f, that is,

$$y_i = f(x_i) \quad \text{for } i = 0, \dots, n,$$

a natural question is how well the interpolating polynomial p approximates f at points $x \neq x_i$.

Before we can formulate an estimate for the error $|f(x) - p(x)|$, we have to introduce the space of k times continuously differentiable functions. Let $[a, b]$ be a closed interval and $k \in \mathbb{N} \cup 0$. Then we denote by $C^k[a, b]$ the space of all functions $f : [a, b] \to \mathbb{R}$ which have a continuous k-th derivative $f^{(k)}$. $C^0[a, b]$ is simply the space of continuous functions. By $C^\infty[a, b]$ we denote the space of infinitely differentiable functions on $[a, b]$. Clearly, if $m \geq k$, then $C^m[a, b] \subset C^k[a, b]$. For $f \in C^0[a, b]$ the following norm

$$\|f\|_\infty = \max_{x \in [a,b]} |f(x)|$$

is always finite.

[2] https://mathworks.com/help/matlab/ref/polyfit.html

Theorem 3.3. *Let $n \in \mathbb{N}$ and $a \leq x_0 < \cdots < x_n \leq b$. Suppose that $f \in C^{n+1}[a,b]$ and that $p \in \mathbb{P}_n$ is the unique interpolating polynomial, that is,*

$$p(x_k) = f(x_k), \quad \text{for } k = 0, \ldots, n.$$

Then, for every $x \in [a,b]$ there is a $\xi \in [a,b]$ such that

$$p(x) - f(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^{n} (x - x_k).$$

Even though this theorem provides us with a formula for the interpolation error at every point $x \in [a,b]$, the problem is that in general we do not know the value of ξ, we only know that it exists. By passing to the maximum, however, we can obtain a pointwise upper bound for the interpolation error

$$|p(x) - f(x)| \leq \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} \prod_{k=0}^{n} |x - x_k|,$$

as well as a normwise upper bound

$$\|p - f\|_\infty \leq \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} \max_{x \in [a,b]} \prod_{k=0}^{n} |x - x_k|. \tag{3.6}$$

These error estimates show that the interpolation error depends on the way the x_i are distributed in $[a,b]$.[3]

The next question we would like to address is this: Can we make the interpolation error $\|p - f\|_\infty$ arbitrarily small by adding more and more interpolation points? The following corollary provides a condition when this is indeed possible. Interestingly, the distribution of interpolation points does not matter here.

Corollary 3.1. *Let $f \in C^\infty[a,b]$ and assume that there exists an $M \geq 0$ such that*

$$\|f^{(n)}\|_\infty \leq M \quad \text{for all } n \in \mathbb{N}. \tag{3.7}$$

For every $n \in \mathbb{N}$ let $a \leq x_0^{(n)} < \cdots < x_n^{(n)} \leq b$ be a set of $n+1$ interpolation points and let $p_n \in \mathbb{P}_n$ be the unique polynomial interpolating f at these points, that is,

$$p_n\big(x_k^{(n)}\big) = f\big(x_k^{(n)}\big), \quad \text{for } k = 0, \ldots, n.$$

Then,

$$\lim_{n \to \infty} \|p_n - f\|_\infty = 0.$$

Proof. Since

$$\prod_{k=0}^{n} |x - x_k^{(n)}| \leq \prod_{k=0}^{n} |b - a| = (b-a)^{n+1},$$

[3] In particular, these estimates suggest looking for interpolation points which minimize $\prod_k |x - x_k|$. This line of reasoning leads to the so-called *Chebyshev nodes*.

we obtain from (3.6)

$$\|p_n - f\|_\infty \leq \frac{\|f^{(n+1)}\|_\infty (b-a)^{n+1}}{(n+1)!} \leq M \frac{(b-a)^{n+1}}{(n+1)!}.$$

The fraction on the right hand side goes to zero as $n \to \infty$, because the factorial $(n+1)!$ grows much faster than the geometric sequence $(b-a)^{n+1}$. □

If the assumption (3.7) is not met, then the approximation quality of the interpolating polynomials can be arbitrarily poor. In fact, there are examples where $\|p_n - f\|_\infty \to \infty$.

Example 3.3 (Runge's phenomenon)**.** Consider the function

$$f(x) = \frac{1}{1+x^2}$$

on the interval $[a, b] = [-5, 5]$. It is infinitely differentiable, but the ∞-norms of its derivatives grow very fast. In particular, there is no number $M \geq 0$ such that $\|f^{(n)}\|_\infty \leq M$ for all $n \in \mathbb{N}$. The polynomials p_n interpolating f at the equidistant points

$$x_k^{(n)} = a + k\frac{b-a}{n},$$

do *not* converge to f, but instead display oscillations near the interval boundaries. In fact the oscillations grow stronger as n increases. This is called *Runge's phenomenon.*

The previous example shows that polynomial interpolation for a large number of equidistant points should be avoided, unless condition (3.7) is satisfied. This condition, however, is extremely restrictive and often not verifiable, because the underlying function f is nonexistent or unknown. But then the following question remains: What to do in practice, if the number of interpolation points is large? There are several options. One of them is discussed in the next example. Another possibility is explored in Section 3.2.

Example 3.4 (Polynomial fitting)**.** Instead of computing the interpolating polynomial $p \in \mathbb{P}_n$ one can try to *fit* a polynomial of degree $m < n$. Picking some basis $\{q_0(x), \ldots, q_m(x)\}$ of $\mathbb{P}_m$, the interpolation conditions in this case lead to the overdetermined system

$$\underbrace{\begin{bmatrix} q_0(x_0) & \cdots & q_m(x_0) \\ \vdots & & \vdots \\ q_0(x_n) & \cdots & q_m(x_n) \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}}_{y},$$

which does not have a solution. However, Section 2.5 offers a remedy: We can at least minimize the norm of the residual $Va - y$ by solving a linear least squares

problem. This way we obtain a polynomial p for which in general $p(x_i) \neq y_i$ but which still has minimal sum of squares

$$\sum_{i=0}^{n} |p(x_i) - y_i|^2$$

among all polynomials in $\mathbb{P}_m$. For a large number of equidistant x_i this approach, with a good choice of m, typically avoids spurious oscillations and therefore produces more useful results than interpolation.

Note that the tall Vandermonde matrix V always has full rank, which is a direct consequence of Lemma 3.1. Therefore, both algorithms discussed in Section 2.5.1 can be used, although for reasons of stability the QR-based approach should be preferred.

Remark 3.2. In Matlab polynomial interpolation/fitting can be done with the command `polyfit`, while `polyval` evaluates a given polynomial at certain points.

3.2 Spline Interpolation

Instead of finding one interpolating polynomial *globally*, that is, for all interpolation points at once, one can divide the interval $[a, b]$ into several subintervals and then interpolate with a low degree polynomial on each subinterval. If the resulting *piecewise* polynomial is also sufficiently smooth at the boundaries of the subintervals, it is called a spline.

The usefulness of splines reaches far beyond the task of interpolation. They are used, for example, in computer-aided design and computer graphics, but also for numerically solving differential equations.

The presentation of the topics covered in this section is along the lines of [7].

Let $\Delta = (x_0, \ldots, x_n)$ be a set of pairwise distinct points in $[a, b]$ such that $x_0 = a$, $x_n = b$, and $x_i < x_{i+1}$ for $0 \leq i \leq n-1$. We call Δ a *partition* of $[a, b]$. A *spline of order* $p \in \mathbb{N}$ is a function $s \in C^{p-1}[a, b]$ such that the restriction of s to each subinterval $[x_i, x_{i+1}]$ is a polynomial of degree $\leq p$, in symbols,

$$s|_{[x_i, x_{i+1}]} \in \mathbb{P}_p \quad \text{for } i = 0, \ldots, n-1.$$

The set of all splines of order p on Δ is denoted by $\mathbb{S}^p(\Delta)$. Clearly, $\mathbb{P}_p \subset \mathbb{S}^p(\Delta)$.

Example 3.5. Consider the function

$$s(x) = \begin{cases} -x^2, & x \leq 0, \\ x^2, & x > 0, \end{cases}$$

and the partition $\Delta = (-1, 0, 1)$. We want to find out whether s is a spline, i.e. whether $f \in \mathbb{S}^p(\Delta)$ for some p. Clearly, s is not in $\mathbb{S}^1(\Delta)$. It is in $\mathbb{S}^2(\Delta)$, however, because its derivative

$$s'(x) = \begin{cases} -2x, & x \leq 0, \\ 2x, & x > 0, \end{cases}$$

is continuous. On the other hand $s''(0)$ does not exist. So $s \notin \mathbb{S}^p(\Delta)$ for $p \geq 3$. Also note that $s' \in \mathbb{S}^1(\Delta)$.

Every linear combination of splines of order p is another spline of order p. In other words $\mathbb{S}^p(\Delta)$ is a vector space.

Lemma 3.2. *Let $n, p \in \mathbb{N}$ and Δ a partition of $[a, b]$ as defined above.*

1. *The set $\mathbb{S}^p(\Delta)$ is a vector space of dimension $n + p$.*
2. *Let $0 \leq k \leq p - 1$. If $s \in \mathbb{S}^p(\Delta)$, then $s^{(k)} \in \mathbb{S}^{p-k}(\Delta)$. That is, the k-th derivative of a p-th order spline is a spline of order $p - k$.*

In these notes we consider interpolation with two types of splines. In Section 3.2.1 we treat the simplest case of *linear splines* ($p = 1$) and afterwards, in Section 3.2.2, we discuss interpolation using *cubic splines* ($p = 3$).

3.2.1 Linear Splines

Let Δ be a partition of $[a, b]$ consisting of $n + 1$ points and $y_0, \ldots, y_n \in \mathbb{R}$. We want to find a linear interpolating spline, that is, a function $s \in \mathbb{S}^1(\Delta)$ satisfying

$$s(x_i) = y_i, \quad \text{for } i = 0, \ldots, n. \tag{3.8}$$

According to Lemma 3.2 the space $\mathbb{S}^1(\Delta)$ has dimension $n + 1$. In other words $n + 1$ conditions are required to uniquely determine a linear spline. On the other hand, the interpolation problem (3.8) imposes just $n+1$ conditions on the unknown spline. This suggests that the linear spline interpolation problem has a unique solution.

Consider the subinterval $[x_i, x_{i+1}]$. On this subinterval we simply have to find the linear interpolating polynomial. We can, for instance, use equation (3.4) to express it in terms of the locally computed Lagrange polynomials

$$\ell_i(x) = y_i \frac{x - x_{i+1}}{x_i - x_{i+1}} + y_{i+1} \frac{x - x_i}{x_{i+1} - x_i}.$$

Having found a formula for all the local interpolants ℓ_i, we can write down the globally interpolating linear spline in the following way

$$s(x) = \begin{cases} \ell_0(x), & x \in [x_0, x_1], \\ \quad \vdots & \\ \ell_{n-1}(x), & x \in [x_{n-1}, x_n]. \end{cases}$$

Another convenient way to express linear splines is in the basis of *hat functions*. The hat function Λ_i, $0 \leq i \leq n$, is defined as the unique linear spline satisfying

$$\Lambda_i(x_j) = \delta_{ij} \quad \text{for } j = 0, \ldots, n.$$

Theorem 3.4. *For a given partition Δ of $[a,b]$ and values $y_0,\dots,y_n \in \mathbb{R}$. The unique interpolating linear spline is given by*

$$s(x) = \sum_{i=0}^{n} y_i \Lambda_i(x).$$

As for polynomials we can ask the following question. Suppose that $y_i = f(x_i)$ for some function $f : [a,b] \to \mathbb{R}$. How well does the piecewise linear interpolant s approximate f? An answer can be derived directly from Theorem 3.3. From now on we denote by

$$h_i = x_i - x_{i-1}, \quad i = 1,\dots,n,$$

the length of the i-th subinterval and by

$$h_{\max} = \max_i h_i$$

the length of the longest subinterval.

Theorem 3.5. *Let $f \in C^2[a,b]$ and denote by $s \in \mathbb{S}^1(\Delta)$ the linear spline that interpolates f at some partition Δ of $[a,b]$. Then*

$$\|s - f\|_\infty \le \frac{\|f''\|_\infty}{8} h_{\max}^2.$$

Proof. Applying Thm. 3.3 to the subinterval $[x_{i-1}, x_i]$ gives the estimate

$$|s(x) - f(x)| \le \frac{|(x-x_i)(x-x_{i-1})|}{2} \max_{y\in[x_{i-1},x_i]} |f''(y)|,$$

which holds for every $x \in [x_{i-1}, x_i]$. The polynomial $(x - x_{i-1})(x - x_i)$ is quadratic and therefore attains its extremum at the midpoint $m = (x_{i-1}+x_i)/2$ of the subinterval

$$|(x-x_{i-1})(x-x_i)| \le |(m-x_{i-1})(m-x_i)| = \frac{h_i^2}{4}.$$

Thus, for every $x \in [x_{i-1}, x_i]$, we have

$$|s(x) - f(x)| \le \frac{h_i^2}{8} \max_{y\in[x_{i-1},x_i]} |f''(y)|.$$

If we now let $x \in [a,b]$ we have to take the maximum over all subintervals in the right hand side:

$$\begin{aligned} |s(x) - f(x)| &\le \max_i \left(\frac{h_i^2}{8} \max_{y\in[x_{i-1},x_i]} |f''(y)| \right) \\ &\le \frac{\|f''\|_\infty}{8} h_{\max}^2. \end{aligned}$$

□

The maximal error between f and s is controlled by the second derivative of f and the length of the longest subinterval. If we steadily decrease $h_{\max}$ by adding more and more interpolation points, the error can be made arbitrarily small. Compare this result to Corollary 3.1 where the interpolation points can be chosen arbitrarily but the assumptions on f are much more restrictive.

3.2.2 Cubic Splines

Now, for a given partition $\Delta = (x_0, \dots, x_n)$ of $[a, b]$ and values $y_0, \dots, y_n \in \mathbb{R}$ we want to find a cubic spline $s \in \mathbb{S}^3(\Delta)$ such that $s(x_i) = y_i$ for all i. Recall Lemma 3.2. If Δ consists of $n + 1$ points, then the space of all cubic splines $\mathbb{S}^3(\Delta)$ has dimension $n + 3$. This suggests that, in contrast to the linear case, a cubic interpolating polynomial will not be uniquely determined by the $n + 1$ interpolation conditions.

Every cubic spline s is a piecewise cubic polynomial that is two times continuously differentiable. According to Lemma 3.2, the second derivative of s is a linear spline and can therefore be expressed in the basis of hat functions

$$s''(x) = \sum_{i=0}^{n} \gamma_i \Lambda_i(x). \tag{3.9}$$

The coefficients γ_i are called the *moments* of the cubic spline s. Below we will make use of the following abbreviation

$$s'_i = s'(x_i).$$

The next lemma gives us an explicit formula for the cubic interpolation spline.

Lemma 3.3. *Assume $s \in \mathbb{S}^3(\Delta)$ satisfies the interpolation conditions*

$$s(x_i) = y_i \quad \textit{for} \quad i = 0, \dots, n.$$

Then, for $x \in [x_{i-1}, x_i]$ we have

$$s(x) = y_i + (x - x_i)s'_i + \gamma_i \frac{(x - x_i)^2}{2} + \frac{\gamma_i - \gamma_{i-1}}{h_i} \frac{(x - x_i)^3}{6}. \tag{3.10}$$

Proof. Let $x \in [x_{i-1}, x_i]$. We express $s(x)$ as a first-order Taylor polynomial around x_i plus remainder in integral form

$$s(x) = y_i + (x - x_i)s'_i + \int_x^{x_i} s''(t)(t - x)\, dt.$$

On $[x_{i-1}, x_i]$ the second derivative s'' is given by

$$s''(t) = \gamma_{i-1}\Lambda_{i-1}(t) + \gamma_i \Lambda_i(t) = \gamma_{i-1} \frac{x_i - t}{x_i - x_{i-1}} + \gamma_i \frac{t - x_{i-1}}{x_i - x_{i-1}},$$

which can be rearranged to

$$= \frac{\gamma_i - \gamma_{i-1}}{h_i}(t - x_i) + \gamma_i.$$

Plugging this into the integral above and evaluating it we get (3.10), which finishes the proof. □

Equation (3.10) completely determines the interpolating spline, as soon as we know the moments γ_i and the values of the first derivative s_i'. The latter can be expressed in terms of the former by evaluating $s(x)$ at $x = x_{i-1}$ according to (3.10)

$$y_{i-1} = y_i - h_i s_i' + h_i^2 \left(\frac{\gamma_i}{3} + \frac{\gamma_{i-1}}{6} \right). \tag{3.11}$$

Now, a simple rearrangement yields a formula for s_i' in which the only unknowns are γ_i and γ_{i-1}. Therefore it remains to determine the moments. For the sake of simplicity we assume from now on that the partition is uniform, that is, all subintervals are of equal length $h_i = h$.

Lemma 3.4. *The vector of moments $\gamma = (\gamma_0, \dots, \gamma_n)^\top$ associated to the interpolating spline $s \in \mathbb{S}^3(\Delta)$ on a uniform partition Δ solves the system $A\gamma = d$, where $A \in \mathbb{R}^{(n-1)\times(n+1)}$ and $d \in \mathbb{R}^{n-1}$ are given by*

$$A = \frac{h}{6}\begin{bmatrix} 1 & 4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \end{bmatrix}, \qquad d = \frac{1}{h}\begin{bmatrix} 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \end{bmatrix}\begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}.$$

Proof. Rewrite (3.11) in the following way

$$-\frac{y_i - y_{i-1}}{h} = -s_i' + \frac{h}{6}(2\gamma_i + \gamma_{i-1}).$$

Now subtract from this the same equation but with i replaced by $i+1$ to obtain

$$\frac{y_{i+1} - y_i}{h} - \frac{y_i - y_{i-1}}{h} = s_{i+1}' - s_i' + \frac{h}{6}(\gamma_{i-1} + \gamma_i - 2\gamma_{i+1}), \tag{3.12}$$

which holds for $1 \le i \le n-1$.

Next we want to get rid of the difference $s_{i+1}' - s_i'$. We first differentiate (3.10) with respect to x yielding

$$s'(x) = s_i' + \gamma_i(x - x_i) + \frac{\gamma_i - \gamma_{i-1}}{h}\frac{(x - x_i)^2}{2}.$$

Since this equation holds for every $x \in [x_{i-1}, x_i]$, we can plug in x_{i-1} for x. After a slight rearrangement of terms we arrive at

$$s_i' - s_{i-1}' = \frac{h}{2}(\gamma_i + \gamma_{i-1}).$$

Now we can replace the difference $s'_{i+1} - s'_i$ in (3.12) and after simplification of the appearing terms we finally end up with

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h} = \frac{h}{6}(\gamma_{i-1} + 4\gamma_i + \gamma_{i+1}).$$

Again this equation holds for $1 \le i \le n-1$. Rewriting all these equations in matrix-vector form we end up with the system $A\gamma = d$ as asserted. □

The system $A\gamma = d$ is underdetermined, as there are $n+1$ unknowns but only $n-1$ equations. Therefore, two more conditions are needed in order to completely determine the moments. This confirms our reasoning from the beginning of this section. Additional conditions are usually imposed on s at the boundary of the interval $[a, b]$. A common choice are the so-called *natural boundary conditions*

$$s''(a) = s''(b) = 0.$$

Cubic splines that satisfy natural boundary conditions are called *natural cubic splines.*

Directly from the definition of moments (3.9) it follows that the moments γ_0 and γ_n of a natural cubic spline must vanish. In this case the system $A\gamma = d$ from Lemma 3.4 simplifies to

$$\frac{h}{6}\begin{bmatrix} 4 & 1 & & \\ 1 & 4 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 4 \end{bmatrix}\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}. \tag{3.13}$$

Theorem 3.6. *Let Δ be a partition of $[a, b]$ and $y_0, \dots, y_n \in \mathbb{R}$. Then there is a unique natural cubic spline $s \in \mathbb{S}^3(\Delta)$ such that $s(x_i) = y_i$ for $i = 0, \dots, n$.*

Proof. The matrix in (3.13) is strictly diagonally dominant, recall Example 2.3, hence regular. Therefore (3.13) uniquely determines the moments, which in turn uniquely determine the interpolating cubic spline s via (3.10) and (3.11). □

For natural cubic splines we have the following error estimate [12].

Theorem 3.7. *Let $f \in C^4[a, b]$ and denote by $s \in \mathbb{S}^3(\Delta)$ the unique natural cubic spline that interpolates f at Δ. Then*

$$\|s - f\|_\infty \le \frac{h_{\max}^5}{h_{\min}} \left\|f^{(4)}\right\|_\infty.$$

For a uniform partition this means

$$\|s - f\|_\infty \le h^4 \left\|f^{(4)}\right\|_\infty.$$

So if we let $h \to 0$ by increasing the number of x_i, then the interpolation error should decrease like h^4. For nonuniform partitions we need an additional assumption to obtain a similar result: As we add more x_i the ratio $h_{\max}/h_{\min}$ must stay bounded. If this condition is met, then the interpolation error should decrease like $h_{\max}^4$.

Remark 3.3. In Matlab and Octave the command `spline` performs cubic spline interpolation.

3.3 Trigonometric Interpolation

In certain applications, such as signal processing, data points frequently display an oscillatory or periodic behaviour of some sort. In this case it can make sense to use trigonometric functions (sines and cosines) for interpolation. Further details on trigonometric interpolation and the fast Fourier transform can be found, for instance, in [14].

A *real trigonometric polynomial* is a function $p : \mathbb{R} \to \mathbb{R}$ of the form

$$p(x) = a_0 + \sum_{k=1}^{N} a_k \cos kx + \sum_{k=1}^{N} b_k \sin kx, \tag{3.14}$$

with coefficients $a_k, b_k \in \mathbb{R}$. *Trigonometric interpolation* consists in finding a trigonometric polynomial satisfying $p(x_j) = y_j$ for given data points

$$(x_0, y_0), \ldots, (x_{n-1}, y_{n-1}) \in \mathbb{R}^2.$$

3.3.1 Reduction to Polynomial Interpolation

Define $2N + 1$ complex coefficients according to

$$\begin{aligned} \gamma_0 &= a_0, \\ \gamma_k &= \frac{a_k - ib_k}{2}, \quad 1 \le k \le N, \\ \gamma_{-k} &= \frac{a_k + ib_k}{2}, \quad 1 \le k \le N. \end{aligned}$$

Now we can use Euler's formula

$$e^{iy} = \cos y + i \sin y, \quad y \in \mathbb{R}, \tag{3.15}$$

to write (3.14) as a *complex* trigonometric polynomial

$$p(x) = \sum_{k=-N}^{N} \gamma_k e^{ikx}.$$

Note that, since $e^{ikx} = \left(e^{ix}\right)^k$, this looks like a polynomial in the complex variable $z = e^{ix}$ except for the fact that k also takes negative values. We can, however, pull out the factor z^{-N} and rewrite

$$p(x) = \sum_{k=-N}^{N} \gamma_k z^k = z^{-N} \sum_{k=0}^{2N} \gamma_{k-N} z^k.$$

Now the interpolation condition $p(x_j) = y_j$ is equivalent to

$$\sum_{k=0}^{2N} \gamma_{k-N} z_j^k = z_j^N y_j, \tag{3.16}$$

where $z_j = e^{ix_j}$. Note that, the left-hand side of (3.16) is a complex polynomial evaluated at z_j. Thus we have found a way to reformulate the trigonometric interpolation problem as a polynomial interpolation problem over $\mathbb{C}$.

It is important to note that the considerations from Section 3.1.1 about polynomial interpolation remain valid in the complex setting. In particular, if we have $n = 2N + 1$ interpolation conditions, then the unique polynomial of degree $\leq 2N$ satisfying (3.16) for $j = 0, \ldots, 2N$ can be written in terms of Lagrange polynomials

$$\sum_{j=0}^{2N} z_j^N y_j L_j(z), \quad \text{where} \quad L_j(z) = \prod_{\substack{k=0 \\ k \neq j}}^{2N} \frac{z - z_k}{z_j - z_k}.$$

It follows that the solution to the trigonometric interpolation problem, i.e. the function we are actually looking for, is given by

$$p(x) = e^{-ixN} \sum_{j=0}^{2N} e^{ix_j N} y_j L_j(e^{ix}) = \sum_{j=0}^{2N} y_j \prod_{\substack{k=0 \\ k \neq j}}^{2N} \frac{\sin \frac{x - x_k}{2}}{\sin \frac{x_j - x_k}{2}}. \tag{3.17}$$

The second equality above is a consequence of trigonometric sum-to-product identities.[4] That (3.17) is indeed a trigonometric polynomial of the form (3.14) can also be verified using trigonometric identities. Thus we have established the following result.

Theorem 3.8. *Let $x_0, \ldots, x_{2N} \in \mathbb{R}$ be pairwise distinct and $y_0, \ldots, y_{2N} \in \mathbb{R}$ be arbitrary. Then there is a unique trigonometric polynomial of the form (3.14) such that $p(x_j) = y_j$ for $j = 0, \ldots, 2N$.*

The practical usefulness of formula (3.17), like that of (3.5), is limited. In Section 3.3.3 we consider a more effective method of solving the trigonometric interpolation problem for equidistant points.

3.3.2 Equidistant Interpolation Points

From now on we restrict our attention to the trigonometric interpolation problem for uniform partitions of the interval $[0, 2\pi]$, i.e. where

$$x_j = \frac{2\pi j}{n}, \quad j = 0, \ldots, n - 1. \tag{3.18}$$

[4] See, e.g., https://en.wikipedia.org/wiki/List_of_trigonometric_identities

In this case, using complex numbers leads to a linear system with a very special structure. Therefore we first look for a *complex* trigonometric polynomial

$$q(x) = \sum_{j=0}^{n-1} c_j e^{ijx} \tag{3.19}$$

with $c_j \in \mathbb{C}$, which interpolates the data points. Once we have found q, we will clarify in Lemma 3.6 how to obtain the real trigonometric polynomial p, i.e. the function we are actually interested in, from q.

Setting

$$w_n = e^{\frac{2\pi i}{n}}$$

we have

$$w_n^k = \begin{cases} e^{ix_k}, & k = 0, \dots, n-1 \\ 1, & k = n. \end{cases}$$

The interpolation conditions for q can now be written as

$$q(x_k) = \sum_{j=0}^{n-1} c_j e^{ijx_k} = \sum_{j=0}^{n-1} c_j w_n^{jk} = y_k, \quad \text{for } k = 0, \dots n-1,$$

or in matrix-vector notation

$$F_n c = y, \tag{3.20}$$

with the *Fourier matrix* $F_n = (w_n^{jk})_{j,k=0}^{n-1} \in \mathbb{C}^{n\times n}$. This system is very easy to solve, once we have collected some properties of F_n.

Lemma 3.5. *The Fourier matrix is symmetric, that is, $F_n^\top = F_n$ and $n^{-\frac{1}{2}} F_n$ is unitary.*

Proof. The matrix F_n is symmetric, because $w_n^{jk} = w_n^{kj}$. For unitarity we have to show

$$\left(n^{-\frac{1}{2}} F_n\right)^* n^{-\frac{1}{2}} F_n = n^{-\frac{1}{2}} F_n \left(n^{-\frac{1}{2}} F_n\right)^* = I,$$

which, due to the symmetry of F_n, is equivalent to

$$\overline{F}_n F_n = F_n \overline{F}_n = nI.$$

Let v_k be the k-th column of F_n. Then the entry in row k and column k of both $\overline{F}_n F_n$ and $F_n \overline{F}_n$ is given by

$$v_k^* v_k = \sum_{j=0}^{n-1} \overline{w_n^{jk}} w_n^{jk} = \sum_{j=0}^{n-1} |w_n^{jk}|^2 = \sum_{j=0}^{n-1} 1 = n.$$

On the other hand, for an off-diagonal entry, we let $\ell \neq k$ and compute

$$v_k^* v_\ell = \sum_{j=0}^{n-1} \overline{w_n^{jk}} w_n^{j\ell} = \sum_{j=0}^{n-1} w_n^{-jk} w_n^{j\ell} = \sum_{j=0}^{n-1} \left(w_n^{(\ell-k)}\right)^j,$$

which is a geometric sum. We use the well-known formula

$$\sum_{j=0}^{n-1} a^j = \frac{1-a^n}{1-a}, \quad a \neq 1,$$

to get

$$v_k^* v_\ell = \frac{1 - w_n^{n(\ell-k)}}{1 - w_n^{\ell-k}} = 0,$$

because $w_n^n = 1$ and $w_n^{\ell-k} \neq 1$. □

Theorem 3.9. *The solution to the trigonometric interpolation problem (3.20), that is, the vector of coefficients $c = (c_0, \ldots, c_{n-1})^\top \in \mathbb{C}^n$ of the interpolating polynomial is given by*

$$c = \frac{1}{n} \overline{F}_n y. \tag{3.21}$$

Proof. Since the matrix F_n is unitary up to a scaling factor, it is in particular invertible. Therefore the solution of (3.20) is given by $c = F_n^{-1} y$. It remains to show that $F_n^{-1} = \frac{1}{n} \bar{F}_n$. From the previous lemma we know that

$$\left(n^{-\frac{1}{2}} F_n\right)^{-1} = \left(n^{-\frac{1}{2}} F_n\right)^* = n^{-\frac{1}{2}} \overline{F}_n^\top = n^{-\frac{1}{2}} \overline{F}_n.$$

Combining this equality with the fact that $\left(n^{-\frac{1}{2}} F_n\right)^{-1} = n^{\frac{1}{2}} F_n^{-1}$ completes the proof. □

Having found the complex interpolating polynomial (3.19), we demonstrate how to obtain its real counterpart (3.14).

Lemma 3.6. *Let $F_n c = y$ and define the following numbers: If n is odd, set*

$$\begin{aligned} N &= (n-1)/2, \\ a_0 &= c_0, \\ a_k &= c_k + c_{n-k}, \quad 1 \leq k \leq N, \\ b_k &= i(c_k - c_{n-k}), \quad 1 \leq k \leq N. \end{aligned} \tag{3.22}$$

If n is even, set

$$\begin{aligned} N &= n/2, \\ a_0 &= c_0, \\ a_k &= c_k + c_{n-k}, \quad 1 \leq k \leq N-1, \\ b_k &= i(c_k - c_{n-k}), \quad 1 \leq k \leq N-1, \\ a_N &= c_N, \\ b_N &= 0. \end{aligned} \tag{3.23}$$

All a_k and b_k defined above are real. The corresponding real trigonometric polynomial p defined via (3.14) satisfies $p(x_j) = y_j$ for $j = 0, \ldots, n-1$.

Proof. That all a_k and b_k are real can be verified by writing (3.21) in a componentwise fashion, that is,

$$c_k = \frac{1}{n}\sum_{j=0}^{n-1} \overline{w_n^{jk}}\, y_j.$$

We leave the details as an exercise.

Next, it follows from Euler's formula (3.15) that

$$\cos y = \frac{e^{iy}+e^{-iy}}{2} \quad \text{and} \quad \sin y = \frac{e^{iy}-e^{-iy}}{2i}.$$

Setting $y = kx_j$ and taking into account that

$$e^{-ikx_j} = e^{-ikx_j}\underbrace{e^{i2\pi j}}_{=1} = e^{-ik2\pi j/n}e^{i2\pi j} = e^{i(n-k)x_j},$$

we find

$$\cos kx_j = \frac{e^{ikx_j}+e^{i(n-k)x_j}}{2} \quad \text{and} \quad \sin kx_j = \frac{e^{ikx_j}-e^{i(n-k)x_j}}{2i}.$$

Using these two identities we can rewrite $p(x_j)$ as

$$\begin{aligned} p(x_j) &= a_0 + \sum_{k=1}^{N} (a_k \cos kx_j + b_k \sin kx_j) \\ &= a_0 + \sum_{k=1}^{N} \left(a_k \frac{e^{ikx_j}+e^{i(n-k)x_j}}{2} + b_k \frac{e^{ikx_j}-e^{i(n-k)x_j}}{2i} \right) \\ &= a_0 + \sum_{k=1}^{N} \left(\frac{a_k - ib_k}{2} e^{ikx_j} + \frac{a_k + ib_k}{2} e^{i(n-k)x_j} \right). \end{aligned} \tag{3.24}$$

Note that the relations

$$a_k = c_k + c_{n-k}, \quad b_k = i(c_k - c_{n-k})$$

from (3.22) and (3.23) are equivalent to

$$c_k = \frac{a_k - ib_k}{2}, \quad c_{n-k} = \frac{a_k + ib_k}{2}.$$

Therefore, if n is odd, we directly obtain from (3.24) that

$$p(x_j) = c_0 + \sum_{k=1}^{N} \left(c_k e^{ikx_j} + c_{n-k} e^{i(n-k)x_j} \right) = \sum_{k=0}^{n-1} c_k e^{ikx_j} = q(x_j) = y_j,$$

which proves the claim. If n is even, we recall from (3.23) that $a_N = c_N$ and $b_N = 0$, and we have

$$\begin{aligned} p(x_j) &= a_0 + \sum_{k=1}^{N-1} \left(\frac{a_k - ib_k}{2} e^{ikx_j} + \frac{a_k + ib_k}{2} e^{i(n-k)x_j} \right) \\ &\quad + \frac{a_N - ib_N}{2} e^{iNx_j} + \frac{a_N + ib_N}{2} e^{i(n-N)x_j} \\ &= c_0 + \sum_{k=1}^{N-1} \left(c_k e^{ikx_j} + c_{n-k} e^{i(n-k)x_j} \right) + c_N e^{iNx_j} \\ &= \sum_{k=0}^{n-1} c_k e^{ikx_j} = q(x_j) = y_j. \end{aligned}$$

□

Remark 3.4. The function p constructed in the previous lemma agrees with q at all interpolation points x_j, but not necessarily in between those points.

Example 3.6. Consider the trigonometric interpolation problem for $n = 4$ and $y = (0, 1, 1, 0)^\top$. We have $w_4 = e^{\frac{\pi i}{2}} = i$ and the Fourier matrix is given by

$$F_4 = \begin{bmatrix} w_4^0 & w_4^0 & w_4^0 & w_4^0 \\ w_4^0 & w_4^1 & w_4^2 & w_4^3 \\ w_4^0 & w_4^2 & w_4^4 & w_4^6 \\ w_4^0 & w_4^3 & w_4^6 & w_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}.$$

From Theorem 3.9 it follows that $c = \frac{1}{4}\overline{F}_4 y = \frac{1}{4}(2, -1 - i, 0, -1 + i)^\top$. Now Lemma 3.6 tells us how to obtain the real trigonometric polynomial that interpolates the four data points. We find $a_0 = \frac{1}{2}$, $a_1 = -\frac{1}{2}$, $a_2 = 0$, $b_1 = \frac{1}{2}$, and $b_2 = 0$. Therefore,

$$p(x) = \frac{1}{2}(1 - \cos x + \sin x)$$

is the solution to the interpolation problem.

3.3.3 Fast Fourier Transform

The function

$$\mathcal{F}_n : \mathbb{C}^n \to \mathbb{C}^n, \quad \mathcal{F}_n(y) := \overline{F}_n y.$$

is called *discrete Fourier transform*[5] in analogy to its continuous counterpart

$$\mathcal{F}f(\xi) = \int_{-\infty}^{\infty} f(x) e^{-ix\xi}\, dx.$$

[5]Note that the literature is not consistent in regards to this definition. Some authors include a factor of $\frac{1}{n}$ or $\frac{1}{\sqrt{n}}$ in the definition of $\mathcal{F}_n$.

The (discrete) Fourier transform is one of the most important concepts in the exact sciences with numerous applications ranging from applied areas such as signal processing to theoretical ones like the analysis of differential equations. Essentially, it transforms a function of space or time (e.g. an audio signal) into one of spatial or temporal frequency. One of the reasons for its usefulness is the fact that many operations, such as convolution or filtering, are much easier to perform in frequency space. Unsurprisingly, the *Fast Fourier Transform* (FFT), is widely considered to be one the most important algorithms [4]. This section loosely follows Chapter 9.3 of [15].

Solving the trigonometric interpolation problem on the points

$$x_j = \frac{2\pi j}{n}, \quad j = 0, \dots, n-1,$$

leads to the $n \times n$ linear system $F_n c = y$. From Sections 2.2 to 2.4 we know that the number of flops required for solving such a problem using a direct method is, in general, cubic in n. According to Thm. 3.9, however, the solution can be found by

1. assembling the matrix $\frac{1}{n}\overline{F}_n$,
2. matrix-vector multiplication $\frac{1}{n}\overline{F}_n y$.

For general matrices and vectors of dimension n the number of flops required for both tasks is quadratic in n, which is already much faster than a general direct method. However, for reasons of periodicity F_n only has n different entries $w_n^0, \dots, w_n^{n-1}$, making the first step linear in n. Moreover, using the FFT the matrix-vector multiplication can be realized with essentially $n \log_2 n$ operations. This is a significant speedup. For example, if $n = 2^{10} = 1024$, then $n \log_2 n \approx 10^4$ while $n^2 \approx 10^6$.

At the heart of the FFT lies, again, a matrix factorization. We start with an example.

Example 3.7. We return to the case of $n = 4$ interpolation points. It turns out that F_4 can be factorized in the following way

$$F_4 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & i \\ 1 & & -1 & \\ & 1 & & -i \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix} \tag{3.25}$$

The matrix on the right is a permutation matrix P. When applied to a vector it puts the entries with even indices ahead of the entries with odd indices. Note, however, that in this section we start indexing with 0 instead of 1. Therefore, the matrix P maps $(c_0, c_1, c_2, c_3)^\top$ to $(c_0, c_2, c_1, c_3)^\top$.

The matrix in the middle, in fact, contains two copies of F_2:

$$\begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} = \begin{bmatrix} w_2^0 & w_2^0 & & \\ w_2^0 & w_2^1 & & \\ & & w_2^0 & w_2^0 \\ & & w_2^0 & w_2^1 \end{bmatrix} = \begin{bmatrix} F_2 & \\ & F_2 \end{bmatrix}.$$

Finally, the leftmost matrix in the factorization (3.25) can be generalized to higher dimensions by realizing that

$$\begin{bmatrix} 1 & & 1 & \\ & 1 & & i \\ 1 & & -1 & \\ & 1 & & -i \end{bmatrix} = \begin{bmatrix} I & D_2 \\ I & -D_2 \end{bmatrix},$$

where I is the 2×2 identity matrix and $D_2 = \operatorname{diag}(1, i) = \operatorname{diag}(w_4^0, w_4^1)$.

Therefore, the following sequence of steps leads to the same result as direct multiplication of F_4 with a vector $c = (c_0, c_1, c_2, c_3)^\top$.

1. Rearrange the entries of c so that the ones with even indices are ahead of those with odd indices.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \mapsto \begin{bmatrix} c_0 \\ c_2 \\ c_1 \\ c_3 \end{bmatrix} =: \begin{bmatrix} c_{\text{even}} \\ c_{\text{odd}} \end{bmatrix}$$

   Here we set $c_{\text{even}} = (c_0, c_2)^\top$ and $c_{\text{odd}} = (c_1, c_3)^\top$.

2. Multiply both the top and bottom halves of the resulting vector with F_2.

$$\begin{bmatrix} c_{\text{even}} \\ c_{\text{odd}} \end{bmatrix} \mapsto \begin{bmatrix} F_2 c_{\text{even}} \\ F_2 c_{\text{odd}} \end{bmatrix}$$

3. Multiply the bottom half of the resulting vector with D_2. Add the result to the top half to form the first two entries of the final result. Subtract it from the top half to form the last two entries of the final vector.

$$\begin{bmatrix} F_2 c_{\text{even}} \\ F_2 c_{\text{odd}} \end{bmatrix} \mapsto \begin{bmatrix} F_2 c_{\text{even}} + D_2 F_2 c_{\text{odd}} \\ F_2 c_{\text{even}} - D_2 F_2 c_{\text{odd}} \end{bmatrix} = F_4 c \tag{3.26}$$

The matrix factorization encountered in this example extends to general even $n \in \mathbb{N}$ in the following way.

Lemma 3.7. *Let $m \in \mathbb{N}$ and $n = 2m$. Then the $n \times n$ Fourier matrix F_n admits the following factorization*

$$F_n = \begin{bmatrix} I & D_m \\ I & -D_m \end{bmatrix} \begin{bmatrix} F_m & \\ & F_m \end{bmatrix} P_n, \tag{3.27}$$

where I is the $m \times m$ identity matrix, $D_m = \operatorname{diag}(w_n^0, \ldots, w_n^{m-1})$ and P_n is the permutation matrix that maps every vector $c = (c_0, \ldots, c_{n-1})^\top \in \mathbb{C}^n$ to

$$P_n c = (c_0, c_2, \ldots, c_{n-2}, c_1, c_3, \ldots, c_{n-1})^\top.$$

If in Lemma 3.7 m is again an even number, $m = 2k$ say, we can factorize both copies of F_m in (3.27) and obtain

$$\begin{bmatrix} F_m & \\ & F_m \end{bmatrix} = \begin{bmatrix} I & D_k & & \\ I & -D_k & & \\ & & I & D_k \\ & & I & -D_k \end{bmatrix} \begin{bmatrix} F_k & & & \\ & F_k & & \\ & & F_k & \\ & & & F_k \end{bmatrix} \begin{bmatrix} P_m & \\ & P_m \end{bmatrix},$$

which yields a new factorization of the original Fourier matrix F_n

$$F_n = \begin{bmatrix} I & D_m \\ I & -D_m \end{bmatrix} \begin{bmatrix} I & D_k & & \\ I & -D_k & & \\ & & I & D_k \\ & & I & -D_k \end{bmatrix} \begin{bmatrix} F_k & & & \\ & F_k & & \\ & & F_k & \\ & & & F_k \end{bmatrix} \begin{bmatrix} P_m & \\ & P_m \end{bmatrix} P_n.$$

More generally, if $n = 2^p$, then we can perform this recursive decomposition p times, and compute the matrix-vector product $F_n c$ accordingly. This is the main idea behind the FFT. The following theorem shows that this approach pays off in terms of number of operations.

Theorem 3.10. *Let $p \in \mathbb{N}$, $n = 2^p$ and $c \in \mathbb{C}^n$. Computation of the product $F_n c$ as outlined above requires $\frac{3}{2} n \log_2 n$ flops.*

Proof. Using the recursive factorization approach the Fourier matrix F_n can be written as a product of $2p + 1$ matrices, p of which are permutation matrices and do not require arithmetic operations. The matrix in the middle will look like

$$\begin{bmatrix} F_1 & & \\ & \ddots & \\ & & F_1 \end{bmatrix},$$

which is just an identity matrix, because $F_1 = 1$. Thus we only need to consider the remaining p matrices on the left hand-side. Multiplication with one of these matrices requires $\frac{n}{2}$ multiplications, because, first, the matrices D_j are diagonal and, second, they are multiplied with only one half of the entries of each vector; recall (3.26). In addition each of the remaining p matrices requires n additions. The total number of flops is therefore

$$p\big(\frac{n}{2} + n\big) = \frac{3}{2} np = \frac{3}{2} n \log_2 n.$$

□

So far we have only discussed how to efficiently multiply with F_n. The reasoning above, however, is easily modified to apply to $\overline{F}_n$ as well by taking the complex conjugate of equation (3.27).

Remark 3.5. In Matlab and Octave the (inverse) Fast Fourier Transform can be called via `fft` and `ifft`.

Chapter 4

Numerical Integration

The aim of numerical integration is to approximate the numerical value of definite integrals. In these notes we restrict our attention to one-dimensional integrals over intervals of finite length. Thus, for $a < b \in \mathbb{R}$ and $f : [a, b] \to \mathbb{R}$, we want to approximate the integral

$$I[f] = \int_a^b f(x)\,dx.$$

Clearly, if F is an antiderivative of f, that is $F'(x) = f(x)$, then $I[f] = F(b) - F(a)$. This raises the question what the purpose of numerical integration is. First, there are functions whose antiderivatives are difficult or impossible to express as elementary functions. (An elementary function is a function that is a finite combination of arithmetic operations, trigonometric functions, exponentials, logarithms, and so on.) An example of a function whose antiderivative is not elementary is $f(x) = e^{-x^2}$. Second, even if an antiderivative of f can be computed, it might just be more efficient to numerically integrate f instead of evaluating F. Finally, in certain situations the integrand might not be known on the whole interval $[a, b]$ but only at a few points. While this makes determining an antiderivative impossible in general, numerical approximation of $I[f]$ might still be feasible.

In order to numerically integrate a given function f we consider *quadrature rules* $Q[f] \approx I[f]$. These will take the following general form

$$Q[f] = \sum_{i=0}^{n} w_i f(x_i)$$

with *weights* $w_i \in \mathbb{R}$ and *nodal points* $x_i \in [a, b]$. That is, $Q[f]$ is a linear combination of a finite number of function values.

Example 4.1 (Midpoint rule)**.** The simplest example of such a quadrature rule is maybe the *midpoint rule*

$$Q[f] = (b - a) f\Big(\frac{a + b}{2}\Big).$$

The midpoint rule approximates the area below the function graph with the area of a rectangle with sides $b-a$ and $f((a+b)/2)$. Therefore it is exact for constant functions $f(x) = c$. However, it is not hard to see that even linear polynomials $f(x) = kx + d$ are integrated exactly by the midpoint rule. Using our notation from Section 3.1, we can formulate this property of the midpoint rule as

$$Q[p] = I[p] \quad \text{for all} \quad p \in \mathbb{P}_1.$$

Example 4.2 (Trapezoidal rule)**.** Another example is the *trapezoidal rule*

$$Q[f] = \frac{b-a}{2} f(a) + \frac{b-a}{2} f(b).$$

Visually it approximates the integral $I[f]$ with the combined area of two rectangles, one with sides $(b-a)/2$ and $f(a)$, and one with $(b-a)/2$ and $f(b)$, or equivalently with that of a trapezoid with vertices $(a,0)$, $(b,0)$, $(a,f(a))$, and $(b,f(b))$. The trapezoidal rule, too, is exact for polynomials of degree one.

In general, we say a quadrature rule Q has *degree* $k \in \mathbb{N}$, if all $p \in \mathbb{P}_k$ are integrated exactly, that is,

$$Q[p] = I[p] \quad \text{for all} \quad p \in \mathbb{P}_k.$$

The next result tells us what maximal degree we can expect of a quadrature rule with $n+1$ nodal points.

Lemma 4.1. *The degree of*

$$Q[f] = \sum_{i=0}^{n} w_i f(x_i)$$

cannot be higher than $2n+1$.

Proof. We prove the assertion by constructing a $p \in \mathbb{P}_{2n+2}$ for which $Q[p] \neq I[p]$. Define

$$p(x) = \prod_{k=0}^{n} (x - x_k)^2,$$

where the x_k are just the nodal points of the quadrature rule. The polynomial p is obviously of degree $2(n+1) = 2n+2$. Since it only consists of quadratic terms it is nonnegative. Hence, its integral must be positive $I[p] > 0$. On the other hand p has been constructed in such a way that $Q[p] = 0$. □

4.1 Newton-Cotes Formulas

Newton-Cotes formulas are a special class of quadrature rules. Their basic idea is to replace the integrand f by the polynomial p that interpolates f at the nodal points x_i, that is, $Q[f] \coloneqq I[p]$. The polynomial p can be integrated exactly by taking the right linear combination of its values at the nodal points.

More specifically, let f be given together with a set of nodal points $x_0, \ldots, x_n$. From Section 3.1 we know that the unique polynomial $p \in \mathbb{P}_n$ satisfying $p(x_i) = f(x_i)$ for $i = 0, \ldots, n$ can be expressed in terms of the Lagrange polynomials

$$p(x) = \sum_{i=0}^{n} f(x_i) L_i(x).$$

The main argument behind Newton-Cotes formulas is that $I[p]$ should be close to $I[f]$, if p is close to f. Therefore

$$\begin{aligned} I[f] \approx I[p] &= \int_a^b p(x)\,dx = \int_a^b \Big(\sum_{i=0}^{n} f(x_i) L_i(x) \Big)\,dx \\ &= \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b L_i(x)\,dx}_{w_i} =: Q[f]. \end{aligned} \tag{4.1}$$

Lemma 4.2. *The degree of a Newton-Cotes formula with $n+1$ nodal points, as defined by (4.1), is at least n.*

Proof. Let $f \in \mathbb{P}_n$. Then the unique polynomial interpolating f is f itself. Therefore $I[f] = Q[f]$. □

If the nodal points are equispaced and such that $x_0 = a$ and $x_n = b$, that is,

$$x_i = a + i\frac{b-a}{n}, \quad \text{for } i = 0, \ldots, n,$$

then we have a so-called *closed Newton-Cotes formula*. The closed Newton-Cotes formula for $n = 1$ is the trapezoidal rule from Example 4.2, which can be shown by computing the two weights w_0 and w_1 according to (4.1).

Example 4.3. The closed Newton-Cotes formula for $n = 2$ is *Simpson's rule*

$$Q[f] = \frac{b-a}{6}\Big[f(a) + 4f\Big(\frac{a+b}{2}\Big) + f(b) \Big].$$

Lemma 4.2 tells us that all quadratic polynomials are integrated exactly by Simpson's rule. However, due to reasons of symmetry, the degree of Simpson's rule is actually 3, which can be seen as follows.

Consider the *cubic* interpolant q of f. In the Newton basis for the points $x_0 = a$, $x_1 = b$, $x_2 = (a+b)/2 =: m$ and arbitrary x_3 it reads

$$q(x) = a_0 + a_1(x-a) + a_2(x-a)(x-b) + a_3(x-a)(x-b)(x-m)$$

for some a_0, a_1, a_2 and a_3. Note that the quadratic interpolant, i.e. the one on which Simpson's formula is based, can be written as

$$p(x) = a_0 + a_1(x-a) + a_2(x-a)(x-b),$$

where the coefficients a_0, a_1, a_2 are the same as for q. (This is a nice property of the Newton basis!) But now

$$\int_a^b (x-a)(x-b)(x-m) = 0,$$

because m is the midpoint of a and b and therefore

$$Q[f] = \int_a^b p(x)\,dx = \int_a^b q(x)\,dx.$$

In particular every cubic polynomial is integrated exactly.

More generally, we have the following.

Theorem 4.1. *A closed Newton-Cotes formula with $n+1$ nodal points has degree*

$$r = \begin{cases} n, & \text{if } n \text{ is odd,} \\ n+1, & \text{if } n \text{ is even.} \end{cases}$$

If a and b are not nodal points, then the resulting Newton-Cotes formula is called *open*. Such quadrature rules can be used for improper integrals like

$$\int_0^1 \frac{dx}{\sqrt{x}},$$

where the integrand cannot be evaluated at one of the endpoints, but the integral is finite nonetheless.

4.1.1 Error estimates

The error estimates for polynomial interpolation can be used to estimate the *quadrature error*

$$|Q[f] - I[f]|$$

of a Newton-Cotes formula Q. An important ingredient for obtaining such quadrature error estimates is the *triangle inequality for integrals*

$$\Big|\int_a^b g(x)\,dx\Big| \le \int_a^b \big|g(x)\big|\,dx.$$

Example 4.4 (Error estimate for the trapezoidal rule)**.** The trapezoidal rule Q as defined in Example 4.2 is the closed Newton-Cotes formula for $n = 1$. This means that

$$Q[f] = \frac{b-a}{2}\,[f(a) + f(b)] = I[p],$$

where p is the linear polynomial that interpolates f at the two points $x_0 = a$ and $x_1 = b$. By Thm. 3.3 we have the following error estimate for p

$$|p(x) - f(x)| \le \frac{\|f''\|_\infty}{2}|x-a||x-b|,$$

for all $x \in [a,b]$. Therefore

$$\begin{aligned} |Q[f] - I[f]| = |I[p] - I[f]| &= \Big| \int_a^b \big(p(x) - f(x)\big)\, dx \Big| \\ &\le \int_a^b \big|p(x) - f(x)\big|\, dx \\ &\le \frac{\|f''\|_\infty}{2} \int_a^b (x-a)(b-x)\, dx \\ &= \frac{\|f''\|_\infty}{12}(b-a)^3. \end{aligned}$$

Example 4.5 (Error estimate for Simpson's rule)**.** Taking into account Example 4.3 a similar derivation shows that Simpson's rule satisfies

$$|Q[f] - I[f]| \le \frac{\|f^{(4)}\|_\infty}{2880}(b-a)^5.$$

Theorem 4.2. *A Newton-Cotes formula Q of degree r satisfies*

$$|Q[f] - I[f]| \le C \frac{\|f^{(r+1)}\|_\infty}{(r+1)!}(b-a)^{r+2} \tag{4.2}$$

for all $f \in C^{r+1}[a,b]$, where the value of the constant $C > 0$ depends on the distribution of the nodal points.

Not surprisingly, Newton-Cotes formulas suffer from the same problems as polynomial interpolation does: A large number of equidistant nodal points should be avoided, unless the derivatives of f can be guaranteed to have small ∞-norms. Fortunately, the very idea that overcomes this problem in the case of interpolation also works for numerical integration.

4.2 Composite Rules

Instead of using Newton-Cotes formulas with a large number of nodal points, it is often better to subdivide the domain of integration $[a,b]$ and use quadrature rules with fewer points on every subinterval. This leads to the definition of a *composite rule* $Q_N[f]$

$$I[f] = \int_a^b f(x)\, dx = \sum_{j=1}^N \int_{a_j}^{b_j} f(x)\, dx \approx \sum_{j=1}^N Q^{(j)}[f] =: Q_N[f],$$

where $a = a_1 < b_1 = a_2 < \cdots < b_{N-1} = a_N < b_N = b$. The following theorem shows how this subdivision strategy affects the quadrature error.

Theorem 4.3. *Let Q_N be a composite rule of length $N \in \mathbb{N}$. Suppose that every subinterval is of equal length $b_j - a_j = \frac{b-a}{N}$ and that $Q^{(j)}$ is a closed Newton-Cotes formula of degree $r \in \mathbb{N}$ for every $j = 1, \dots, N$. Then,*

$$|Q_N[f] - I[f]| \leq C \frac{\|f^{(r+1)}\|_\infty}{(r+1)!} \frac{(b-a)^{r+2}}{N^{r+1}}, \tag{4.3}$$

for every $f \in C^{r+1}[a,b]$.

Proof. Applying (4.2) to every subinterval we get

$$\begin{aligned}
|Q_N[f] - I[f]| &= \Big| \sum_{j=1}^{N} Q^{(j)}[f] - \int_a^b f(x)\,dx \Big| \\
&= \Big| \sum_{j=1}^{N} \Big(Q^{(j)}[f] - \int_{a_j}^{b_j} f(x)\,dx \Big) \Big| \\
&\leq \sum_{j=1}^{N} \Big| Q^{(j)}[f] - \int_{a_j}^{b_j} f(x)\,dx \Big| \\
&\overset{(4.2)}{\leq} \sum_{j=1}^{N} C \frac{(b_j - a_j)^{r+2}}{(r+1)!} \max_{x \in [a_j, b_j]} \Big| f^{(r+1)}(x) \Big| \\
&= \frac{C}{(r+1)!} \sum_{j=1}^{N} \frac{(b-a)^{r+2}}{N^{r+2}} \max_{x \in [a_j, b_j]} \Big| f^{(r+1)}(x) \Big| \\
&\leq C \frac{(b-a)^{r+2}}{(r+1)! N^{r+2}} \sum_{j=1}^{N} \|f^{(r+1)}\|_\infty \\
&= C \frac{\|f^{(r+1)}\|_\infty}{(r+1)!} \frac{(b-a)^{r+2}}{N^{r+1}}.
\end{aligned}$$

□

Note that the quadrature error of the composite rule (4.3) is equal to the error of the non-composite Newton-Cotes formula (4.2) divided by N^{r+1}. We can state the result of Thm. 4.3 more succinctly by saying that for every $f \in C^{r+1}[a,b]$ there is a $B \geq 0$ such that

$$|Q_N[f] - I[f]| \leq B \Big(\frac{1}{N} \Big)^{r+1}. \tag{4.4}$$

Since B is independent of N, we can decrease the quadrature error arbitrarily much by simply increasing the number of subintervals N. When a composite rule Q_N satisfies estimate (4.4), we say that Q_N *converges with order* $r+1$.

The trapezoidal rule, for instance, has degree $r = n = 1$. Therefore, by Thm. 4.3 the composite trapezoidal rule converges with order $r + 1 = 2$. Simpson's rule has degree $r = n + 1 = 3$. The order of convergence of the composite Simpson's rule is therefore $r + 1 = 4$.

4.3 Gauss Quadrature

Lemma 4.1 states that the degree of a quadrature rule with $n+1$ nodal points cannot be higher than $2n+1$. This raises the question whether it can actually be equal to $2n+1$.

Example 4.6 (Two-point Gauss-Legendre rule[1])**.** Let $[a,b] = [-1,1]$ and consider a general quadrature rule with $n+1 = 2$ nodal points

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(x_0) + w_1 f(x_1). \tag{4.5}$$

We want to find values for the weights $w_0, w_1 \in \mathbb{R}$ and nodes $x_0, x_1 \in [-1,1]$ such that (4.5) becomes an equality for all cubic polynomials $f \in \mathbb{P}_3$. If we plug in the four monomials $1, x, x^2, x^3$, we get a system of four nonlinear equations in four unknowns

$$\begin{aligned} w_0 + w_1 &= 2 \\ w_0 x_0 + w_1 x_1 &= 0 \\ w_0 x_0^2 + w_1 x_1^2 &= \frac{2}{3} \\ w_0 x_0^3 + w_1 x_1^3 &= 0. \end{aligned}$$

Subtracting x_0^2 times the second equation from the fourth yields

$$w_0 x_0^3 + w_1 x_1^3 - w_0 x_0^3 - w_1 x_1 x_0^2 = w_1 x_1 (x_1^2 - x_0^2) = 0.$$

The weight w_1 cannot equal zero, since then Q would be a 1-point rule. Similarly, $x_1 = 0$ is impossible. In this case the second equation ($x_0 w_0 = 0$) would again imply a 1-point rule. Therefore $0 \neq x_0 = -x_1$ must hold. Plugging this into the second equation gives

$$w_0 x_0 + w_1 x_1 = w_0 x_0 - w_1 x_0 = x_0 (w_0 - w_1) = 0,$$

which implies $w_0 = w_1$. From the first equation we now get $w_0 = w_1 = 1$. Finally, the third equation yields $x_0 = -\frac{1}{\sqrt{3}}$ and $x_1 = \frac{1}{\sqrt{3}}$.

Thus, we have found the so-called *two-point Gauss-Legendre rule*

$$Q[f] = f\Big(\frac{-1}{\sqrt{3}}\Big) + f\Big(\frac{1}{\sqrt{3}}\Big),$$

which is exact for all polynomials of degree less than or equal to 3.

In general, the $(n+1)$*-point Gauss-Legendre rule*

$$Q[f] = \sum_{i=0}^{n} w_i f(x_i)$$

[1] This example is taken from [9].

has maximal degree $2n + 1$. Its weights and nodal points solve the $2n + 2$ nonlinear equations

$$\sum_{i=0}^{n} w_i x_i^k = \int_{-1}^{1} x^k \, dx = \frac{1 - (-1)^{k+1}}{k+1}, \quad k = 0, \ldots, 2n+1.$$

As above, the weights and nodes are usually computed for the interval $[-1, 1]$ and tables containing their precise values can be found in the literature.[2] If the domain of integration happens to be different from $[-1, 1]$, one can transform the integral first using integration by substitution:

$$\int_a^b f(x) \, dx = \frac{b-a}{2} \int_{-1}^{1} f\left(\frac{a+b}{2} + x\frac{b-a}{2}\right) dx.$$

For further reading on numerical integration see, for instance, [12, 14].

Remark 4.1. In Matlab and Octave there are several commands for numerically integrating a given function. If a functional expression is available for your integrand, the command `integral` chooses a suitable method for integration. If only data points are available, then e.g. `trapz` can be used for numerical integration using the composite trapezoidal rule.

[2] See, for instance, the standard reference [1, Tab. 25.4] or its digital successor [11, §3.5(v)].

Chapter 5
Eigenvalue Problems and SVD

In this chapter we will consider a matrix $A \in \mathbb{C}^{n\times n}$ and numerically find its eigenvalues and eigenvectors. We use the more general case of complex matrices because even for real matrices eigenvalues and eigenvectors can be complex valued.

A standard book for numerical treatment of eigenvalue problems is [13].

Remember that matrices are representations of linear maps/transformations, like projections, rotations, reflections, translation, scaling, permutation and all combinations of them. Eigenvalue problems help to decompose such transformations into basic components (cf. principal component analysis). In practice, eigenvalues and eigenvectors are used to discover modes of vibrations (acoustics, (quantum) mechanics), e.g. instabilities of structures can be detected. They also play an important role in the convergence analysis of iterative schemes, long term behavior and stability of dynamical systems and (stability) analysis of numerical methods for differential equations.

5.1 Mathematical background

For a given matrix A we search for pairs of *eigenvalues* $\lambda \in \mathbb{C}$ and *eigenvectors* $x \in \mathbb{C}^n$ that satisfy

$$Ax = \lambda x \quad x \in \mathbb{C}^n \setminus \{0\},\ \lambda \in \mathbb{C}\ ,$$

where we will sometimes use the abbreviation (x, λ) for the *eigenpair*.
We could find the eigenvalues by the roots λ_i of the *characteristic polynomial*

$$\chi(z) = \det(A - zI) = c(z - \lambda_1)(z - \lambda_2)\dots(z - \lambda_n)\ ,$$

which is a non-linear problem that is, however, too complex and instable for practical purpose. We define the *algebraic multiplicity* of an eigenvalue λ to be its multiplicity as a root of $\chi(z)$.

For each eigenvalue λ we can calculate the corresponding eigenvectors by solving the linear equation

$$(A - \lambda I)\, v_j = 0\ .$$

The linear space spanned by the eigenvectors v_j is called the *eigenspace* and is denoted with E_λ. The number of linear independent eigenvectors (that is the dimension of E_λ) is called the *geometric multiplicity* of the eigenvalue λ.

Example 5.1. Consider the two matrices

$$A = \begin{bmatrix} 2 & & \\ & 2 & \\ & & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 1 & \\ & 2 & 1 \\ & & 2 \end{bmatrix}.$$

What is the characteristic polynomial, eigenvalues, algebraic and geometric multiplicities?

An eigenvalue whose algebraic multiplicity is greater than its geometric multiplicity is called a *defective eigenvalue*. A matrix with at least one defective eigenvalue is called a *defective matrix*.

We call the matrices A and B *similar* if there exists a non-singular $X \in \mathbb{C}^{n\times n}$ such that $B = XAX^{-1}$. Similar matrices have the same characteristic polynomial, eigenvalues, and algebraic and geometric multiplicities.

Numerical algorithms to find eigenvalues are **not** based on finding roots of the characteristic polynomial, since computation of the characteristic polynomial is too expensive and it would also be a highly ill-conditioned problem (algorithm would be unstable). Eigenvalues are computed by iteration methods (e.g. power iteration, see Section 5.2.1) or by eigenvalue revealing factorizations (see Section 5.2.3).

Lemma 5.1. *Some important matrix factorizations are:*

1. *If A is non-defective, then a diagonalization $A = X\Lambda X^{-1}$ exists, where Λ is a diagonal matrix.*
2. *If A is normal (i.e. $AA^* = A^*A$) a unitary diagonalization $A = Q\Lambda Q^*$ exists, where Q is a unitary matrix (i.e. $Q^* = Q^{-1}$), and Λ is a diagonal matrix. Note that all hermitian matrices are normal.*
3. *A unitary triangulation (*Schur factorization*) $A = QTQ^*$ always exists, where Q is unitary and T is upper triangular.*

The following properties hold:

- Is a matrix positive definite, then all eigenvalues are real and positive.
- Is a matrix hermitian (i.e. $A = A^*$), then all eigenvalues are real.
- Is λ an eigenvalue of the nonsingular matrix A with eigenvector x, then $\frac{1}{\lambda}$ is an eigenvalue of A^{-1} with eigenvector x.
- A and A^* have the same set of eigenvalues.

Typical applications of eigenvalues are situations where we are interested in $A^k = X\Lambda^k X^{-1}$ or $e^A = Xe^\Lambda X^{-1}$ for a non-defective matrix A. The second expression is especially useful for analyzing systems of linear differential equations.

5.1.1 Estimation of eigenvalues

For a quick estimation of the eigenvalues we can use

Theorem 5.1. (Gershgorin circle theorem) *Given a matrix $A = [a_{ij}] \in \mathbb{C}^{n\times n}$, then for every eigenvalue λ of A, holds*

$$\lambda \in \bigcup_{i=1}^{n} \mathcal{K}_i,$$

with the Gershgorin discs

$$\mathcal{K}_i := \{\zeta \in \mathbb{C} : |\zeta - a_{ii}| \leq \sum_{j\neq i} |a_{ij}|\}.$$

Proof. Let $Ax = \lambda x$ with $x = [x_i] \neq 0$. Then there exists an index i, so that $|x_j| \leq |x_i|$ for all $j \neq i$. $(Ax)_i$ denotes the i-th component of Ax, then

$$\lambda x_i = (Ax)_i = \sum_{j=1}^{n} a_{ij}x_j$$

and therefore,

$$|\lambda - a_{ii}| = \left|\sum_{j\neq i} a_{ij}\frac{x_j}{x_i}\right| \leq \sum_{j\neq i} |a_{ij}| \ .$$

Therefore, $\lambda \in \mathcal{K}_i \subseteq \bigcup_{j=1}^{n} \mathcal{K}_j$. □

Example 5.2 (Exercise)**.** What are the Gershgorin discs of

$$A = \begin{bmatrix} 4 & 0 & -3 \\ 0 & -1 & 1 \\ -1 & 1 & 0 \end{bmatrix} ?$$

5.1.2 The Rayleigh quotient

We can further confine the range in which the eigenvalues lie, when using the *Rayleigh quotient*.

Definition 5.1 (Rayleigh Quotient)**.** For a hermitian matrix A the Rayleigh quotient of a vector x is given by

$$r(x) := \frac{x^*Ax}{x^*x}, \quad x \in \mathbb{C}^n\backslash\{0\}. \tag{5.1}$$

Note that for an eigenvalue-eigenvector pair (λ, v) we have $r(v) = \lambda$, and for each vector x the Rayleigh quotient gives the value $\alpha := r(x)$, which acts most like an eigenvalue. In fact, if we look at the normal equations related to $\|Ax - \alpha x\|_2$ (with Ax the r.h.s. and x the matrix) we get $\alpha x^*x = x^*Ax$ with

the solution $\alpha = r(x)$ for $x \neq 0$.
Further, if we assume a vector w close to an eigenvector v (e.g. w is some approximation to the unknown v) we can estimate the eigenvalue of v by $r(w)$. In fact we have

Lemma 5.2. *Let A be hermitian and $r(x)$ defined as in (5.1). Then the eigenvectors $v_1, v_2, \ldots, v_n$ are stationary points of $r(x)$ and the corresponding eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of A are the stationary values. Further there holds*

$$r(x) - r(v_i) = \mathcal{O}(\|v_i - x\|^2) \quad \text{as } x \to v_i. \tag{5.2}$$

Proof. Application of the quotient-rule yields

$$\nabla r(x) = \frac{2}{x^* x}(Ax - r(x)x).$$

Hence, $\nabla r(v_i) = 0$ with $r(v_i) = \lambda_i$. Exploiting Taylor expansion of r around the eigenvectors v_i gives

$$r(x) = r(v_i) + (x - v_i)^* \nabla r(v_i) + \mathcal{O}(\|v_i - x\|^2). \tag{5.3}$$

Since $\nabla r(v_i) = 0$ this shows (5.2). □

The range of $r(x)$ (for general A not necessarily hermitian)

$$\mathcal{W}(A) := \left\{ \frac{x^* A x}{x^* x} : x \in \mathbb{C}^n \backslash \{0\} \right\} \subseteq \mathbb{C}\,.$$

is called the *numerical range* of the matrix A. Notably, the eigenvalues of A lie in the numerical range of A.

Further properties of $\mathcal{W}(\mathcal{A})$ are

1. $\mathcal{W}(A)$ is connected.
2. If A is hermitian, then $\mathcal{W}(A)$ is the real interval $[\lambda_{\min}, \lambda_{\max}]$. (Courant-Fischer)
3. If A is *skew-symmetric* (i.e. $A^* = -A$), then $\mathcal{W}(A)$ is an imaginary interval, i.e. the convex hull ($\subseteq \mathbb{C}$) of all eigenvalues of A.

Every matrix $A \in \mathbb{C}^{n \times n}$ can be split into an hermitian part (first term) and a skew symmetric part (second term):

$$A = \frac{A + A^*}{2} + \frac{A - A^*}{2}.$$

From this splitting and the above properties we directly derive:

Theorem 5.2. (Theorem of Bendixon)

$$\sigma(A) \subseteq \mathcal{W}\left(\frac{A+A^*}{2}\right) \oplus \mathcal{W}\left(\frac{A-A^*}{2}\right),$$

where $\sigma(A)$ is the spectrum of $A \in \mathbb{C}^{n\times n}$ (i.e. the set containing all eigenvalues of A).

Example 5.3. We use the theorem to further confine the range of Example 5.2. First we calculate the hermitian and skew-symmetric part

$$H = \frac{A+A^*}{2} = \begin{bmatrix} 4 & 0 & -2 \\ 0 & -1 & 1 \\ -2 & 1 & 0 \end{bmatrix}, \quad S = \frac{A-A^*}{2} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The spectra of H and S can be estimated by the theorem of Gerschgorin: This yields [1]

$$\mathcal{R} = [-3, 6] \times [-i, i] \supset \mathcal{W}(A) \supset \sigma(A).$$

The spectrum of A must lie in the intersection of $\mathcal{R}$ and the Gerschgorin discs of Example 5.2. The actual spectrum of A is

$$\sigma(A) = \{-1.7878, 0.1198, 4.6679\}.$$

5.2 Numerical treatment of eigenvalue problems

5.2.1 Power iteration

In this section we numerically calculate eigenvalues and eigenvectors by a method called *power iteration*. For simplicity, we only consider symmetric real matrices. Such matrices are diagonalizable and their eigenvectors form an orthonormal basis. Furthermore, we sort the absolute values of the eigenvalues in descending order $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n| \geq 0$.

We consider the sequence $z_{k+1} = Az_k/\|Az_k\|$. Under certain assumptions the sequence converges to the eigenvector corresponding to the largest eigenvalue of A. We will impose the condition $|\lambda_1| > |\lambda_k|$, $k = 2, \ldots, n$. The approach is outlined in Algorithm 1.

The estimation for the eigenvalue can be found by applying the Rayleigh quotient

$$\lambda_k := \frac{{z_k}^* A z_k}{z_k^* z_k}.$$

We analyze the power iteration by writing the initial guess z_0 as a linear combination of the orthonormal eigenvectors q_i:

$$z_0 = a_1 q_1 + a_2 q_2 + \cdots + a_n q_n \tag{5.4}$$

[1] For H: $\mathcal{K}_1 : |\lambda - 4| \leq 2$, $\mathcal{K}_2 : |\lambda + 1| \leq 1$, $\mathcal{K}_3 : |\lambda| \leq 3$; For S: $\mathcal{K}_1 = \mathcal{K}_3 : |\lambda| \leq 1$, $\mathcal{K}_2 : \lambda = 0$.

Algorithm 1: Power iteration

Data: Matrix A, initial vector z_0 with $\|z_0\| = 1$
Result: Approximation z to eigenvector of A corr. to largest eigenvalue.
Initialization: Set $k \leftarrow 0$
while *Convergence criterion not satisfied* **do**

$$\tilde{z}_{k+1} \leftarrow A z_k \qquad \text{apply } A$$
$$z_{k+1} \leftarrow \frac{\tilde{z}_{k+1}}{\|\tilde{z}_{k+1}\|} \qquad \text{normalize}$$
$$k \leftarrow k + 1$$

end
$z \leftarrow z_k$

The value $\tilde{z}_k$ is a multiple of $A^k z_0$, therefore we write

$$\tilde{z}_k = c_k A^k z_0 \ ,$$

where c_k is some scalar constant (containing normalization factors). Note that $Aq_i = \lambda_i q_i$, thus inserting Eqn. (5.4) yields

$$\begin{aligned} \tilde{z}_k &= c_k(a_1\lambda_1^k q_1 + a_2\lambda_2^k q_2 + \cdots + a_n\lambda_n^k q_n) \\ \tilde{z}_k &= c_k\lambda_1^k(a_1 q_1 + a_2(\lambda_2/\lambda_1)^k q_2 + \cdots + a_n(\lambda_n/\lambda_1)^k q_n) \end{aligned}$$

If k becomes large, the terms $(\lambda_j/\lambda_1)^k$ for $j = 2 \ldots n$ become small (tend to zero, note that $|\lambda_1| > |\lambda_j|$), and therefore $\tilde{z}_k \approx (c_k\lambda_1^k a_1)q_1$ for large iteration number k. Thus, for all z_0, where $a_1 \neq 0$ the proposed Algorithm 1 will converge to the (unit length) eigenvector corresponding to the largest eigenvalue λ_1.

The algorithm is very simple, but of limited use, since it only finds the eigenvector corresponding to the largest eigenvalue, and if λ_1 is not much larger than λ_2 convergence is very slow.
Anyhow, there is a 'famous' example where the power iteration is used, namely the computation of the *Page-Rank*[2] used by Google to sort search results:

Example 5.4 (PageRank - Web graph/linking analysis)**.** Imagine three webpages X,Y and Z. X links to Y and Z, Y to X and Z to Y. We want to find out the importance of the webpages in order to sort them. Webpage X transfers half of its importance to webpage Y and Z. Webpage Y transfers its importance to page X and Z to Y. Thus, the linking matrix looks

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 0 & 0 \end{pmatrix}.$$

This matrix multiplied with an 'importance vector' or 'rank vector' (sum of entries normalized to 1) means that each webpage collects the importance which

[2]Named after Larry Page (Google founder).

it gets from other webpages (incoming links), e.g. if the initial rank vector is $v = (1/3, 1/3, 1/3)^*$ we get

$$(Av)_1 = 1 \cdot \tfrac{1}{3} = \tfrac{1}{3}, \tag{5.5}$$
$$(Av)_2 = \tfrac{1}{2} \cdot \tfrac{1}{3} + 1 \cdot \tfrac{1}{3} = \tfrac{1}{2} \tag{5.6}$$
$$(Av)_3 = \tfrac{1}{2} \cdot \tfrac{1}{3} = \tfrac{1}{6}. \tag{5.7}$$

This can be defined now as an updated rank vector. Iterating this gives the power iteration, where the final vector is called the PageRank vector. The PageRank vector components corresponding to a webpage reflects the probability that a random surfer lands on that webpage by clicking on a link. In fact, this is an example for a *Markov chain*. In practice, the random surfer model is extended by the following model

$$M = (1-p)A + p\,\texttt{ones(n)}/n, \tag{5.8}$$

where $p \in (0,1)$ is the probability that the random surfer will leave the 'clicking-modus' and arbitrarily 'teleport' to a different webpage. All webpages in the latter case are weighted equally, so it is modeled by the matrix with all entries equal to 1 divided by the number of webpages in the linking graph. Mathematically this model is rigorous since the resulting matrix M is positive and column stochastic (column entries sum up to 1). According to the Perron-Frobenius theorem, 1 is an eigenvalue for such matrices with algebraic multiplicity one and it is the largest eigenvalue (absolute values). There is a unique eigenvector (with all entries positive) to the eigenvalue 1 which is column normalized. For the PageRank model (5.8) this eigenvector is the PageRank vector v_*

$$v_* = Mv_*, \tag{5.9}$$

and can be computed with the power iteration. For the value $p = 0.15$ the PageRank vector of the above example is $v_* = (0.3878, 0.3974, 0.2148)^*$. For more details see e.g. [17]. □

5.2.2 Inverse and Rayleigh quotient iteration

We can improve the Power iteration using the following idea: For any $\mu \in \mathbb{R}$ that is not an eigenvalue of A, the eigenvectors of A are the same as of $(A-\mu I)^{-1}$ and if λ is an eigenvalue of A then $(\lambda-\mu)^{-1}$ is an eigenvalue of $(A-\mu I)^{-1}$. [3]

Consequently, if we choose a value μ that is close to an eigenvalue λ of A, then $(\lambda-\mu)^{-1}$ will be large, and therefore the eigenvector of $(A-\mu I)^{-1}$ can be computed very fast by power iteration. This leads to the method called *inverse iteration* which is described in Algorithm 2. The algorithm computes the eigenvector corresponding to the eigenvalue nearest to μ.

We can further improve the inverse iteration by successively improving the eigenvalue estimate μ in each step to increase the rate of convergence. For that

[3] $Av = \lambda v \Leftrightarrow (A-\mu I)v = (\lambda-\mu)v \Leftrightarrow (\lambda-\mu)^{-1}v = (A-\mu I)^{-1}v$.

Algorithm 2: Inverse iteration

Data: Matrix A, initial μ close to the desired eigenvalue, initial vector z_0 with $\|z_0\| = 1$
Result: Approximation z to eigenvector of A corr. to eigenvalue nearest to μ.
Initialization: Set $k \leftarrow 0$
while *Convergence criterion not satisfied* **do**

Solve $(A - \mu I)w = z_k$ for w	"apply" $(A - \mu I)^{-1}$
$z_{k+1} \leftarrow \frac{w}{\|w\|}$	normalize
$k \leftarrow k + 1$	

end
$z \leftarrow z_k$

purpose, we use the Rayleigh quotient

$$r(z) := \frac{z^* A z}{z^* z}, \quad z \in \mathbb{C}^n \backslash \{0\}$$

to estimate the eigenvalue λ from the estimated (normalized) eigenvector z, which minimizes $\|Ax - \lambda x\|$. This leads to the *Rayleigh quotient iteration* outlined in Algorithm 3. This algorithm is very fast and ideally shows a cubic convergence rate.

Algorithm 3: Rayleigh quotient iteration

Data: Matrix A, initial vector z_0 with $\|z_0\| = 1$
Result: Approximation z to an eigenvector of A.
Initialization: Set $k \leftarrow 0$, let $\lambda_0 \leftarrow z_0^* A z_0 / (z_0^* z_0)$
while *Convergence criterion not satisfied* **do**

solve $(A - \lambda_k I)w = z_k$ for w	"apply" $(A - \lambda_k I)^{-1}$
$z_{k+1} := \frac{w}{\|w\|}$	normalize
$\lambda_{k+1} := (z_{k+1})^* A z_{k+1} / (z_{k+1}^* z_{k+1})$	Rayleigh quotient
$k \leftarrow k + 1$	

end
$z \leftarrow z_k$

5.2.3 QR algorithm

The QR algorithm is a stable and simple procedure for calculating all eigenvalues and eigenvectors. The algorithm uses the QR factorization[4] and the next iterate is a recombination of the factors in reverse order (see Algorithm 4).

[4] Not suitable for large-scale problems.

First we observe that since $A_k = Q_k R_k$, we get with $R_k = Q_k^* A_k$

$$A_{k+1} = R_k Q_k = Q_k^* A_k Q_k. \tag{5.10}$$

Thus, A_k and A_{k+1} are similar and if A_k converges, the limit matrix has the same eigenvalues as the initial matrix $A_0 = A$ and the eigenvectors can be seen from the columns of the product (unitary) matrix $Q = Q_0 Q_1 Q_2 \cdots$.

Secondly, the QR algorithm can be seen as a more sophisticated variation of the power iteration (Algorithm 1). Instead of using one single vector, the QR algorithm works with a complete orthonormal basis of vectors, using the QR decomposition for orthogonalization. For a hermitian matrix A the algorithm converges to a diagonal matrix of eigenvalues. The matrix $Q = Q_0 \cdots Q_{k-1} Q_k$ is orthogonal, and the columns of Q are (approximations of) the eigenvectors of A. For the non-hermitian case, the algorithm converges to the (real) Schur form of the matrix (see Lemma 5.1), where A has a block diagonal and blocks can be of size one (real eigenvalues) or size two (complex conjugate eigenvalues). In the latter case a block looks like $\begin{pmatrix} \text{a} & \text{- b} \\ \text{b} & \text{a} \end{pmatrix}$ corresponding to the complex conjugate eigenvalues $a \pm ib$.

Algorithm 4: QR algorithm

Data: Matrix A
Result: Matrix A with eigenvalues in diagonal and corr. eigenvalues in columns of orthogonal matrix Q.
Initialization: Set $k = 0$, $A_0 = A$
while *Convergence criterion not satisfied* **do**
  $Q_k R_k \leftarrow A_k$ QR factorization of A_k
  $A_{k+1} \leftarrow R_k Q_k$ recombine in reverse order
  $k \leftarrow k + 1$
end
$A \leftarrow A_k$
$Q \leftarrow Q_0 \cdots Q_{k-1} Q_k$

The QR algorithm is the standard algorithm for computing all eigenvalues of a matrix. The algorithm can achieve cubic convergence for hermitian matrices, given the following enhancements:

1. Initially, A is reduced to tridiagonal form using Householder transformations.

2. Instead of A_k, the shifted matrix $A_k - \mu_k I$ is factored, where μ_k is an eigenvalue estimate (same idea as Rayleigh quotient iteration, Algorithm 3).

3. If possible A_k is split into sub-matrices (a divide and conquer strategy).

5.2.4 Deflation and dimension reduction

Let us assume real symmetric A. In principle, one could compute the eigenvector x_1 (normalized w.r.t. $\|.\|_2$) corr. to the largest eigenvalue λ_1 (absolute values) by power iteration and apply *deflation* like

$$\widetilde{A} = A - \lambda_1 x_1 x_1^*, \tag{5.11}$$

which is a matrix that has the same eigenvalues and eigenvectors as A except that the largest eigenvalue λ_1 is shifted to zero. In fact, since two different eigenvectors corr. to different nonzero eigenvalues are orthogonal, $\widetilde{A}x = Ax = \lambda x$ for $x \neq x_1$ an eigenvector of A with eigenvalue λ and furthermore, $\widetilde{A}x_1 = 0$. The eigenvector corr. to the second largest eigenvalue could now be computed by using $\widetilde{A}$. For diagonalizable but not necessarily symmetric matrices a similar deflation strategy can be obtained using biorthonormal left- and right eigenvectors, the reader is referred to literature about *Hotelling and Wielandt* deflation techniques.
An interesting variant of deflation is *matrix reduction*. Suppose the deflated matrix

$$\widetilde{A}_{(j)} = A - x/x^{(j)}(e_j^* A), \tag{5.12}$$

where x is an eigenvector of A with j-th component $x^{(j)}$ and $e_j^* A$ is the j-th row of A. Observe, that the $j-$row of the deflated matrix $\widetilde{A}_{(j)}$ contains only zeros, in fact

$$e_j^* \widetilde{A}_{(j)} = e_j^* A - (e_j^* x)/x^{(j)}(e_j^* A) = 0. \tag{5.13}$$

We further suppose orthonormal left and right eigenvectors, that is $y_\ell^* x_m = \delta_{\ell m}$ for left eigenvectors y_ℓ and right eigenvectors x_m. [5] Now multiply $\widetilde{A}_{(j)}$ with a left eigenvector y which is perpendicular to x, that is

$$y^* \widetilde{A}_{(j)} = y^* A - \underbrace{y^* x}_{=0}/x^{(j)}(e_j^* A). \tag{5.14}$$

Since left and right eigenvectors have the same eigenvalues, $\widetilde{A}_{(j)}$ has the eigenvalues of A except that corr. to x. Overall, one can delete the j-th row and column of $\widetilde{A}_{(j)}$ and calculate the eigenvalues of the resulting *reduced matrix* of dimension reduced by one.
There is a *block* approach of the described procedure called *nullspace reduction*, which simultaneously reduces dimensions corr. to several known eigenpairs.

[5] Left eigenvectors are defined by $y^* A = \lambda y^*$ or equivalently by the notion of right eigenvectors of the transposed matrix, i.e., $A^* y = \lambda y^*$. Since $\det(A^* - \lambda I) = \det\big((A - \lambda I)^*\big) = \det(A - \lambda I)$ we have that left and right eigenvalues coincide. Further, if we have $Ax = \lambda x$ and $y^* A = \mu y^*$ with $\lambda \neq \mu$, we have $y^* x = 0$, since (w.l.o.g. $\lambda, \mu \neq 0$, otherwise $y^* x = 0$ immediately) $y^* x = \frac{1}{\mu}(\mu y^*)x = \frac{1}{\mu} y^* A x = \frac{\lambda}{\mu} y^* x$.

5.2.5 Eigenvalues via Arnoldi/Lanczos iteration

Some of the eigenvalues of A (typically including dominant eigenvalues) can be approximated by Ritz values (see Appendix. B. The Hessenberg matrix H_m (resp. T_m in the symmetric case) is computed by Arnoldi (resp. Lanczos) iteration for an arbitrary r_0 (e.g. randomly chosen). Afterwards eigenvalues of the $m \times m$ matrix H_m (resp. T_m) are computed via e.g. the QR algorithm. Further note that if $m = n$, then V_m in $H_m = V_m^* A V_m$ (resp. $T_m = V_m^* A V_m$ in the hermitian case) is unitary and $A = V_m H_m V_m^*$ (resp. $A = V_m T_m V_m^*$ in the hermitian case). In this case an eigenpair (x, λ) of H_m (resp. T_m) translates to an eigenpair $(V_m x, \lambda)$ of A.

Example 5.5. As an example we look at the matrix from the finite difference discretization of the Poisson equation in 2d with $N = n^2 = 100$: The matrix $A \in \mathbb{R}^{N \times N}$ with $N = n^2$ (K_{2d} matrix) is sparse (see Figure 5.1) and can be generated by so-called Kronecker products $A = K_{2d} := K_{1d} \otimes I + I \otimes K_{1d} \in \mathbb{R}^{N \times N}$, where $K_{1d} \in \mathbb{R}^{n \times n}$ is the tridiagonal matrix arising from the central difference quotient, i.e., $K_{1d} = \text{tridiag}(-1, 2, -1)$ and $I \in \mathbb{R}^{n \times n}$ the identity (diag(1)).

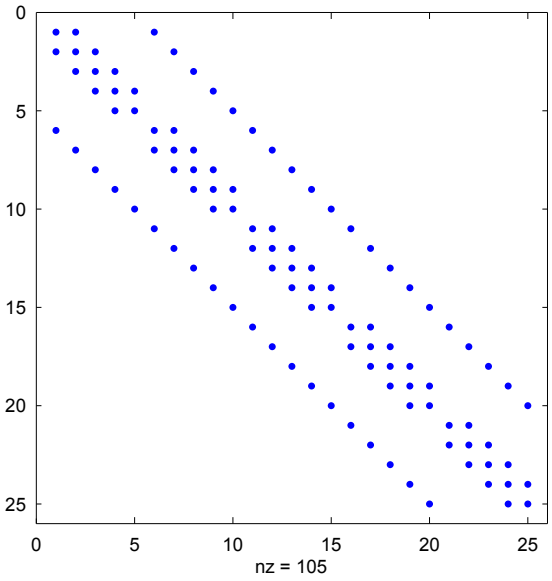

Figure 5.1: Sparsity pattern of K_{2d} matrix

We choose $m = 5$ and $r_0 = (1, 1, \ldots, 1)^*$. Lanczos yields (here represented

with first 4 digits, true calculations used double precision)

$$T_5 = \begin{bmatrix} 0.4000 & 0.5657 & & & \\ 0.5657 & 2.2500 & 1.1214 & & \\ & 1.1214 & 2.7456 & 1.5370 & \\ & & 1.5370 & 3.9276 & 1.8468 \\ & & & 1.8468 & 3.9558 \end{bmatrix}.$$

The eigenvalues of T_5 are

$$\{0.1659, 0.8915, 2.2559, 3.7648, 6.2009\}.$$

True eigenvalues of A which comes closest to the eigenvalues of T_5 are

$$\{0.1620, 0.7713, 2.3383, 3.7635, 6.2036\}.$$

By increasing the Krylov subspace dimension m one can observe better approximations. For instance, choosing $m = 20$, the largest five eigenvalues of T_{20} are

$$\{5.6617, 6.5133, 7.3120, 7.3650, 7.8213\},$$

compared to true eigenvalues of A (which are closest to these Ritz values), which are

$$\{5.6617, 6.5133, 7.2287, 7.3650, 7.8380\}.$$

□

Ritz pairs (estimates for eigenvalues and corr. eigenvectors) can be efficiently computed by deflation techniques for restarted Arnoldi iteration. For a stable implementation compare with Matlab's `eigs` function which is a large scale implementation which also takes only the action on the matrix ($x \mapsto Ax$) as input.

5.3 Generalized eigenvalue problem

Definition 5.2. The problem of finding a vector $v \in \mathbb{C}^n \setminus \{0\}$ so that

$$Av = \lambda Bv \tag{5.15}$$

for $A, B \in \mathbb{C}^{n\times n}$ is called a *generalized eigenvalue problem* and the *generalized eigenvalue* $\lambda \in \mathbb{C}$ obeys the equation

$$\det(A - \lambda_i B) = 0.$$

If B is non-singular, the solution can be obtained by solving the equivalent standard eigenvalue problem

$$B^{-1}Av = \lambda v. \tag{5.16}$$

However, inversion of B might be too expensive and also important properties like symmetry can be lost.
Let us now assume that both A and B are real and symmetric and B is also positive definite. In this case we can define a generalization to the Rayleigh quotient as

$$r(x) := \frac{x^*Ax}{x^*Bx}. \tag{5.17}$$

To see the relation to (5.15) we look at the gradient of (5.17)

$$\begin{aligned}\nabla r(x) =& \frac{2Ax(x^*Bx) - 2(x^*Ax)Bx}{(x^*Bx)^2} = \frac{2Ax(x^*Bx) - 2r(x)(x^*Bx)Bx}{(x^*Bx)^2} \\ =& \frac{2Ax - 2r(x)Bx}{x^*Bx}\end{aligned} \tag{5.18}$$

and set it to zero to find the stationary points

$$\nabla r(x) = 0 \quad \Rightarrow Ax = r(x)Bx. \tag{5.19}$$

Thus, the extreme points of the generalized Rayleigh quotient are attained by the eigenvectors of the generalized eigenvalue problem. Note that for the above derivation the symmetry of A and B, as well as the positive definiteness of B was used. A further consequence of the symmetry is that two eigenvectors v_i, v_j of the generalized problem corresponding to different eigenvalues λ_i and λ_j, resp., are B-orthogonal.
For A and B symmetric and B positive definite one can simultaneously diagonalize such that [18]

$$V^*AV = \Lambda \tag{5.20}$$
$$V^*BV = I, \tag{5.21}$$

where Λ is diagonal and contains the eigenvalues and V contains the eigenvectors (but is not orthogonal $[V^{-1} \neq V^*]$ but invertible [basis of eigenvectors]). In fact, right-multiplication of the second equation with Λ implies now

$$V^*BV\Lambda = \Lambda = V^*AV, \tag{5.22}$$

and hence

$$AV = BV\Lambda. \tag{5.23}$$

This is (5.15), where V contains the eigenvectors as columns and Λ the diagonal matrix with the eigenvalues in its diagonal.

5.4 Singular value decomposition

Theorem 5.3 (Singular Value Decomposition)**.** *Let $A \in \mathbb{C}^{m\times n}$ with rank$(A) = p \leq \min\{m, n\}$, then there exists a unique decomposition called* singular value

decomposition *(SVD) such that*

$$A = U\Sigma V^*, \tag{5.24}$$

where $U := [u_1, \ldots, u_m] \in \mathbb{C}^{m\times m}$ *is unitary* ($U^*U = UU^* = I$), *and* $V := [v_1, \ldots, v_n] \in \mathbb{C}^{n\times n}$ *is unitary, with* $\{u_j\}_{j=1}^m$ *and* $\{v_j\}_{j=1}^n$ *being orthonormal basis sets on* $\mathbb{C}^m$ *and* $\mathbb{C}^n$, *denoted as left and right singular vectors, respectively. Further, the matrix* $\Sigma \in \mathbb{R}^{m\times n}$ *is diagonal*

$$\Sigma := \left[\begin{array}{ccc|c} \sigma_1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & \sigma_p & 0 \\ \hline 0 & \cdots & 0 & 0 \end{array}\right],$$

with real values $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p > 0$, *with* $p \leq \min\{n, m\}$, *called the singular values of* A.

Proof. Can be found in any standard textbook about (numerical) linear algebra, and works e.g. by using the spectral theorem of A^*A. □

Eqn. (5.24) can also be written in the *compact form*

$$A = \sum_{j=1}^{p} \sigma_j u_j v_j^*, \tag{5.25}$$

which is in matrix form

$$A = U_p \Sigma_p V_p^*, \tag{5.26}$$

with $U_p \in \mathbb{C}^{m\times p}$, $V_p \in \mathbb{C}^{n\times p}$ and $\Sigma_p \in \mathbb{R}^{p\times p}$.
The so-called *economy sized* or *thin SVD* is obtained by ignoring columns in the computation of U resp. V depending on whether $m > n$ or $m < n$. In the case $m > n$ we get

$$A = U_n \Sigma_n V^*, \tag{5.27}$$

with $U_n \in \mathbb{C}^{m\times n}$ and $\Sigma_n \in \mathbb{R}^{n\times n}$, where in the case $m < n$ we get

$$A = U\Sigma_m V_m^*, \tag{5.28}$$

with $V_m \in \mathbb{C}^{n\times m}$ and $\Sigma_m \in \mathbb{R}^{m\times m}$.

A geometrical interpretation of the SVD is that the image of a unit sphere under any $m \times n$ matrix is a hyperellipse, where the vectors $\sigma_i u_i$ are the principal semiaxes.

Note that from (5.24)

$$Av_j = \sigma_j u_j \quad \text{for} \quad j = 1 \dots p \,, \tag{5.29}$$

and

$$U^* A V = \Sigma.$$

Also, we have $A^* = V\Sigma U^*$ and hence $A^*U = V\Sigma$, i.e.,

$$A^* u_j = \sigma_j v_j \quad \text{for} \quad j = 1 \dots p \,. \tag{5.30}$$

The SVD shows that all matrices are diagonal for the proper choice of basis sets for the domain and range spaces.

There are fundamental differences between singular values and eigenvalue decompositions (compare with Lemma 5.1):

- SVD uses two different bases, eigenvalue decomposition just one.
- In a SVD the two bases are always orthogonal, in eigenvalue decomposition this is generally not the case (e.g. true if A is normal).
- Not all matrices have an eigenvalue decomposition, but all (even rectangular) matrices have a SVD.

Theorem 5.4. *Nonzero singular values of A are the square roots of the nonzero eigenvalues of A^*A and AA^*.*

Ad proof. This can be seen from $A^*A = V\Sigma^*U^*U\Sigma V^* = V(\Sigma^*\Sigma)V^*$ resp. $AA^* = U(\Sigma\Sigma^*)U^*$. □

It follows:

Theorem 5.5. *If A is hermitian (i.e. $A = A^*$) the singular values of A are the absolute values of the eigenvalues of A.* □

Remark 5.1. If A is hermitian and positive (semi) definite (in general: normal and pos. semi definite), then the SVD coincides with its spectral decomposition $A = Q\Lambda Q^*$, where Λ is the diagonal matrix containing the eigenvalues (real and positive) in descending order and Q the unitary matrix containing the corresponding eigenvectors columnwise. □

Thus, numerical computation of singular values could be based on calculating the eigenvalues of A^*A (or AA^*). However, this method is unstable, since the condition number of A^*A might be much bigger than that of A. [6]

[6] In fact, one defines the condition number for a rectangular rank-p matrix A as $cond_2(A) := \sigma_1/\sigma_p$ and there holds for hermitian A: $cond_2(A^*A) = cond_2(AA^*) = cond_2(A)^2$.

Stable SVD algorithms are based on finding the eigenvalues of the self-adjoint block matrix

$$M = \begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix},$$

which are the singular values of A with positive and negative sign: The equations $AV = U\Sigma$ and $A^*U = V\Sigma$ imply

$$\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix} \begin{pmatrix} V & V \\ U & -U \end{pmatrix} = \begin{pmatrix} V & V \\ U & -U \end{pmatrix} \begin{pmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{pmatrix},$$

which amounts to an eigenvalue decomposition of M.

SVD algorithms scale typically cubically (asymptotically n^3 or mn^2 operations).

5.4.1 Low-rank matrices and Eckart-Young theorem

The problem to find a matrix approximation $\widetilde{A}$ to a rank-p matrix A with smaller (or equal) rank $r \leq p$ is solved by the SVD. In fact, there holds:

Theorem 5.6 (Eckart-Young)**.** *Given the SVD $A = U\Sigma V^*$, the optimization problem*

$$\min_{\widetilde{A}} \|\widetilde{A} - A\|_F \quad \textit{with rank}(\widetilde{A}) = r, \tag{5.31}$$

turns out to have the solution

$$A_r = U\Sigma' V^* = \sum_{j=1}^{r} \sigma_j u_j v_j^*, \tag{5.32}$$

where Σ' is a modification of Σ that contains only the largest r singular values and the others are replaced by zeros. This "truncated SVD" is the unique solution to the above optimization problem iff $\sigma_r \neq \sigma_{r+1}$. For the error there holds

$$\|A - A_r\|_F := \|vec(A - A_r)\|_2 = \sqrt{\sum_{j=r+1}^{p} \sigma_j^2}. \tag{5.33}$$

Proof. Can be found in any standard textbook about (numerical) linear algebra. □

The described approach of discarding small singular values in the approximation of A is often referred to as *truncated SVD*. It can be written in the (tensor-) product form

$$A_r = \sum_{j=1}^{r} \sigma_j \, u_j \, v_j^*. \tag{5.34}$$

A_r can be seen as a compressed version of A with less storage requirements. Matrix-vector multiplication can be performed efficiently

$$A_r x = \sum_{j=1}^{r} \sigma_j (v_j^* x)\, u_j. \tag{5.35}$$

5.4.2 Pseudoinverse and Linear least squares

Imagine an overdetermined system of linear equations $Ax = b$, where $A \in \mathbb{R}^{m\times n},\ m > n$. The solution to the linear least squares problem

$$\min \|Ax - b\|_2^2 \tag{5.36}$$

with *minimum Euclidean norm*[7] is given by

$$x_* = A^+ b = V\Sigma^+ U^* b, \tag{5.37}$$

where $A^+ = V\Sigma^+ U^*$ is the (Moore-Penrose) pseudoinverse of A and obtained from the SVD[8] $A = U\Sigma V^*$ with Σ^+ containing the reciprocal (non-zero) singular values. This method is computationally more expensive than methods based on other factorizations like QR.[9] However, it can be combined with the truncated SVD approach if very small singular values would introduce too much noise (ill-conditioned problem). The condition number of a linear least squares problem is given as $cond_2(A) := \|A\|_2 \|A^+\|_2 = \sigma_1/\sigma_p$ and therefore decreased by discarding very small singular values.

[7] The so-called minimum norm solution which is relevant in the rank deficient case.

[8] Since often $m \gg n$ it is advisable to compute only the "eco-SVD", i.e., only the first n left singular vectors.

[9] SVD is more stable than QR but slower.

Chapter 6

Nonlinear systems of equations

In this chapter we solve the equation $f(x) = 0$ numerically for general (nonlinear) f in one ($f : \mathbb{R} \to \mathbb{R}$) and higher dimensions ($f : \mathbb{R}^n \to \mathbb{R}^n$). This will be achieved by methods that are based on the *Newton's method*. We mention, however, that there are also ways to numerically solve systems of nonlinear equations which do not rely on derivative information of f, so-called *derivative free methods*. Some of them will be discussed in the excerise course as simple programming examples.

Obviously, solving such equations has many applications. A frequent application arises from (unconstrained) optimization of functions ($\mathbb{R}^n \to \mathbb{R}$), since the derivative/gradient of the function can be set to zero in order to find extremal points.

A good book which covers numerical optimization and also nonlinear equations is Nocedal/Wright [10].

6.1 Newton's method

Newton's method is the classical local root finding scheme. It is also sometimes referred to as *Newton-Raphson* scheme. Newton's method relies on first order derivative information. We first motivate the scheme for general systems of equations by the one-dimensional version, which has a simple geometric illustration (Fig. 6.1).

6.1.1 One-dimensional geometric motivation

We want to numerically find the root (zero) of a function $f : \mathbb{R} \to \mathbb{R}$, that is, we search for $x \in \mathbb{R}$ so that $f(x) = 0$. Furthermore, we assume f to be continuously differentiable.

We start with an initial guess x_0 and iteratively improve it. The idea of the method is to locally approximate the function by its tangent around the guess

x_k. In fact, Taylor expansion gives

$$f(x_k + \Delta x) = f(x_k) + \Delta x f'(x_k) + O\big((\Delta x)^2\big). \tag{6.1}$$

In the next step the x-intercept of the tangent is computed ($0 = f(x_k) + \Delta x f'(x_k)$). The value $x_{k+1} := x_k + \Delta x$ will typically be a better approximation. Thus we obtain the following iterative method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \tag{6.2}$$

In this way the guess x_k might be improved in every step (illustrated in Figure 6.1), but we state that it could also fail, e.g. if the guess is not close enough to the root.

Newton's method is frequently used in solving equations as well as in optimization. It is a very powerful approach since its convergence rate is up to quadratic. [1] However, the Newton method has two main difficulties:

- First, if a stationary point of the function f is encountered, i.e., $f'(x_k) = 0$, we cannot calculate Eqn (6.2).
- Second, Newton's method does not globally converge. Indeed, for arbitrary initialization x_{init} we can not expect Newton's method to converge. It is a *local method*.

One possibility for obtaining a higher probability of convergence (i.e. towards a globally convergent scheme) is the combination of Newton's method with one of the line search algorithms of Section 6.3.

Example 6.1. Consider $f(x) = x\cos x + x^2$ with derivative $f'(x) = \cos x - x\sin x + 2x$. Tab. 6.1 shows the first 5 iterates starting with $x_0 = 1$ converging to $x_* = 0$. Note, that the function f also has a second root $x_*^{(2)} \approx -0.7391$ which is attracted by the choice of (e.g.) $x_0 = -1$.

Note the quadratic convergence (in the end). □

6.1.2 Q-convergence rates

Suppose a (nonlinear) iteration given by $x_{k+1} = \Phi_f(x_k)$ that approximates a solution x_* of the system of nonlinear equations $f(x) = 0$, that is $x_* = \Phi_f(x_*)$.

Definition 6.1. Let $e_k := \|x_* - x_k\|$ the error of the k-th iterate and $e_k \to 0$ as $k \to \infty$.

[1] It is *Q-superlinear*: The new error $|x_{k+1} - x_*| \le r_k |x_k - x_*|$ where $r_k \to 0$ for $k \to 0$. For certain regularity assumptions on f around the solution, convergence is even *Q-quadratic*: $|x_{k+1} - x_*| \le M|x_k - x_*|^2$ for some $M > 0$, i.e. the new error is expected to be in the order of the squared previous error, at least asymptotically.

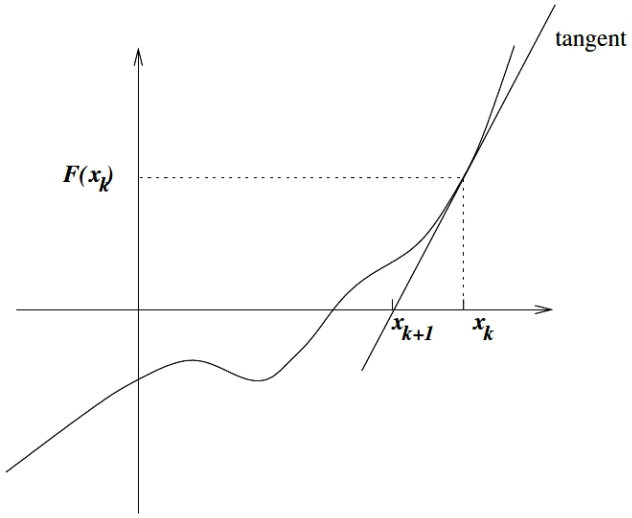

Figure 6.1: One iteration of the one-dimensional Newton method. Fig. taken from [10].

Table 6.1: Example 6.1 of root-finding Newton's iteration.

k	x_k
0	1
1	9.3E-02
2	6.7E-03
3	4.4E-05
4	1.9E-09
5	3.8E-18

- The convergence is *Q-linear* if $e_k/e_{k-1} \leq r \in (0,1)$ for all k sufficiently large.
- The convergence is *Q-superlinear* if $e_k/e_{k-1} \to 0$ as $k \to \infty$.
- The convergence is *Q-quadratic* if $e_k/e_{k-1}^2 \leq M(>0)$ for all k sufficiently large.

□

It is easy to see that Q-superlinear convergence implies Q-linear convergence and Q-quadratic convergence implies Q-superlinear convergence. The opposite is not true which can be seen from counter examples.

6.1.3 Higher-dimensional generalization

In this section we consider the case where we have n equations and n unknowns. Thus for a given function $f : \mathbb{R}^n \to \mathbb{R}^n$ we search for $x \in \mathbb{R}^n$ so that $f(x) = 0$. Again, we assume f to be continuously differentiable.

We use the same concept as in the one-dimensional case. We first linearize around the point x_k:

$$f(x_k + d_k) = f(x_k) + J_f(x_k)d_k + \mathcal{O}(\|d_k\|^2), \tag{6.3}$$

with $x_k \in \mathbb{R}^n$ and $d_k \in \mathbb{R}^n$, and where $J_f(x_k) \in \mathbb{R}^{n\times n}$ is the Jacobian matrix.[2]

As in the one-dimensional case we set the linearized equation to zero, i.e. $f(x_k) + J_f(x_k)d_k = 0$, and solve for the *Newton direction* d_k, thus

$$d_k = -J_f^{-1}(x_k)f(x_k), \tag{6.4}$$

where $J_f^{-1}(x_k)$ is the inverse of the Jacobian matrix. This yields the following iteration

$$x_{k+1} = x_k - J_f^{-1}(x_k)f(x_k). \tag{6.5}$$

The equation corresponds to the one-dimensional case (see Eqn. (6.2)), but the method is **not** implemented in this way. Since the calculation of the inverse Jacobian is computationally expensive, d_k is calculated from the linear system

$$\begin{aligned} J_f(x_k)d_k &= -f(x_k)\ , \\ x_{k+1} &= x_k + d_k. \end{aligned} \tag{6.6}$$

This method is described in Algorithm 5.

Decomposition methods for the computation of the update d_k need order of n^3 operations. Iterative methods may often be faster depending (e.g.) on some known structure or sparsity of the Jacobian. Also remember that iterative methods only need the action of the matrix, i.e. $x \mapsto Ax$, where the realization of the product Ax can be a 'black-box'.

Under certain assumptions convergence of Newton's method is Q-superlinear or even Q-quadratic.

Theorem 6.1 (local convergence, [10])**.** *Suppose that f is continuously differentiable in a convex open set $\mathcal{D} \subset \mathbb{R}^n$. Let $x_* \in \mathcal{D}$ be a nondegenerate solution of $f(x) = 0$ (that is $J_f(x_*)$ is nonsingular), and let $\{x_k\}$ be the sequence of iterates generated by Newton's method (Alg. 5). Then, if x_k is sufficiently close to x_*, the method converges Q-superlinearly. If f is in addition also Lipschitz continuously differentiable, convergence is even Q-quadratic.* □

Example 6.2. Let $f(x, y) = (\cos x - \sin y, xy - \pi/4)^T$ where $f(x, y) = 0$ has a non-degenerate solution at $x_* = (0.39876, 1.96956)^T$. We use Newton's method with $x_0 = (2, 2)^T$. Tab. 6.2 shows errors in $\|.\|_2$ and ratios of successive errors. We observe Q-superlinear convergence. □

[2] The Jacobian matrix contains the first derivatives.

Algorithm 5: Newton's method.

Data: Function $f\colon \mathbb{R}^n \to \mathbb{R}^n$, initial guess $x_0 \in \mathbb{R}^n$
Result: Approximate root x
Initialization: Set $k \leftarrow 0$;
while *Convergence criterion not satisfied* **do**

$$\begin{aligned} \text{Solve } J_f(x_k)\, d_k &= -f(x_k) \text{ for } d_k \\ x_{k+1} &\leftarrow x_k + d_k \\ k &\leftarrow k+1 \end{aligned}$$

end
$x \leftarrow x_k$

Table 6.2: Example 6.2 of root-finding Newton's iteration.

k	e_k	e_k/e_{k-1}	e_k/e_{k-1}^2
0	1.6015	-	-
1	0.1206	0.0753	0.0470
2	0.0080	0.0661	0.5481
3	0.0001	0.0081	1.0117
4	0.0000	0.0001	0.9551
5	0.0000	0.0000	14.2320

Example 6.3. Let $f(x,y) = (xy, x-y)^T$ where $f(x,y) = 0$ has a degenerate solution at $x_* = (0,0)^T$. We use Newton's method with $x_0 = (1,1)^T$. Tab. 6.3 shows errors in $\|.\|_2$ and ratios of successive errors. We do not observe Q-

Table 6.3: Example 6.3 of root-finding Newton's iteration.

k	e_k	e_k/e_{k-1}	e_k/e_{k-1}^2
0	1.4142	-	-
1	0.7071	0.5	0.3536
2	0.3536	0.5	0.7071
3	0.1768	0.5	1.4142
4	0.0884	0.5	2.8284
5	0.0442	0.5	5.6569
6	0.0221	0.5	11.3137

superlinear convergence, in fact, the sequence $e_k/e_{k-1} \equiv 1/2$ and $e_k/e_{k-1}^2 = \frac{1}{\sqrt{2}}\, 2^{k-2} \to \infty$. □

The Newton method in higher dimensions has similar drawbacks as the one dimensional method:

(i) Problems will arise when the matrix $J_f(x_k)$ is ill-conditioned, because in this case an accurate computation of the Newton direction as approximate solution to the equation $J_f(x_k)d = -f(x_k)$ is difficult. Especially, the *Q-superlinear* convergence can not be guaranteed in the case of a degenerate solution x_*, that is $J_f(x_*)$ is singular.

(ii) We can not guarantee convergence for arbitrary initial values - it is a *local method.*

(iii) Additionally, the computation of the Jacobian $J_f(x_k)$ itself (each Newton step) can be computationally expensive for large systems of equations. In fact, often the Jacobian is not even known, only its action or an approximation - often only as 'black box'. Iterative schemes might be the only choice - preconditioning might be problematic (Jacobian changes each iteration).

In the following two sections we discuss methods that have been developed specifically to counter these problems.

6.2 Inexact Newton methods

Inexact Newton methods approximate the Jacobian. Often they are referred to as *quasi-Newton methods*, where this term is especially common for the application of approximate Newton methods in optimization.

6.2.1 One-dimensional motivation: Secant method

We consider a function $f : \mathbb{R} \to \mathbb{R}$ and search for $x \in \mathbb{R}$ so that $f(x) = 0$. As in the one-dimensional Newton method the idea is to employ an iterative algorithm, defining the next iterate x_{k+1} by approximating the function f around x_k linearly by $f(x) \approx f(x_k) + f'(x_k)(x - x_k)$, and then solving the linear equation $f(x_k) + f'(x_k)(x - x_k) = 0$ for x.

Since $f'(x_k)$ could be difficult to calculate numerically, we avoid direct calculation, and use the following approximation instead:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}. \tag{6.7}$$

Note that the approximation becomes more exact as the method converges (i.e. $x_k - x_{k-1} \to 0$).

Inserting this additional approximation into Eqn.(6.2) yields

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k). \tag{6.8}$$

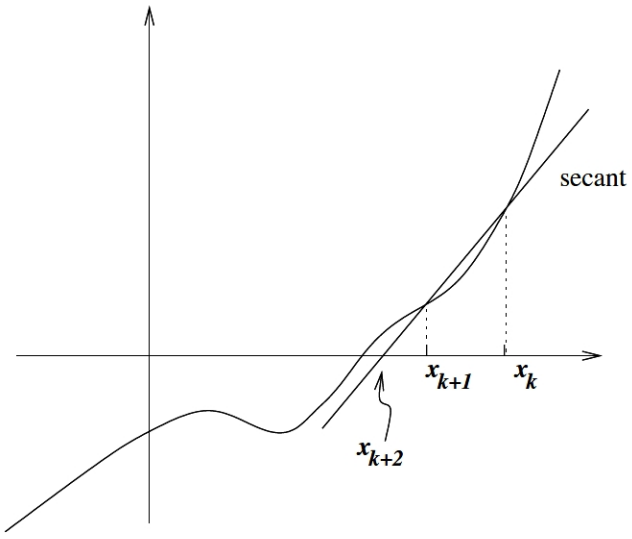

Figure 6.2: One iteration of the one-dimensional secant method. Taken from [10].

More intuitively, the idea behind this approach is to approximate the function f near x_k by the secant through the points $\big(x_k, f(x_k)\big)$ and $\big(x_{k-1}, f(x_{k-1})\big)$. So, again a linear local model is used but the 'exact' linear model (tangent) is replaced by the secant approximation. Hence, this approach is referred to as *secant method*. See also Fig. 6.2 for an illustration.

Note, that the secant method has an interpretation that is very similar to that of Newton's method: In Newton's method, we have approximated the function f by its linearization around x_k. Here, we use the following approximation $\tilde{f}$ of f satisfying $\tilde{f}(x_k) = f(x_k)$ and the one-dimensional *secant equation*

$$\tilde{f}'(x_k)(x_k - x_{k-1}) = f(x_k) - f(x_{k-1}) \; . \tag{6.9}$$

This is the key equation to higher dimensional extensions of the method.

6.2.2 Higher-dimensional generalization

For generalizing the one-dimensional secant method to higher dimensions we first start analogously to Section 6.1.3 and linearize the function $f : \mathbb{R}^n \to \mathbb{R}^n$ around $x_k \in \mathbb{R}^n$ by

$$\tilde{f}(x_k + d_k) = f(x_k) + B_k d_k$$

where $B_k \in \mathbb{R}^{n \times n}$ is an approximation of the Jacobian $J_f(x_k)$. We calculate d_k from $\tilde{f}(x_k + d_k) = 0$, which can be computed by solving the system of linear equations $B_k d_k = -f(x_k)$.

This approach makes sense, if the following conditions are satisfied:

- The matrix B_k must be a reasonable good approximation of $J_f(x_k)$ in some sense. Otherwise, we cannot expect good convergence rates.
- The approximation B_k should be rather cheap.
- The equation $B_k d_k = -f(x_k)$ can be solved without too much effort. This could be achieved by approximating B_k^{-1} instead of B_k.

Inexact-Newton methods choose the estimates B_k of the Jacobian $J_f(x_k)$ to fit the *secant equation* for higher dimensions, which is given by

$$B_k(x_k - x_{k-1}) = f(x_k) - f(x_{k-1}). \tag{6.10}$$

For more than one dimension this system of equations is highly under-determined (n equations, n^2 unknowns). As a result the choice of B_k is not unique.

Broyden's method uses an initial estimate B_k and improves it by taking the solution of the secant equation which is a minimal modification to B_k in the following sense:

We assume that the new estimate B_{k+1} is close to the previous estimate B_k with respect to the Frobenius norm (thus minimize $\|B_{k+1} - B_k\|_F$). As we know from the Eckart-Young theorem discussed in Sec. 5.4, this can be achieved by a rank-1 update.

More precisely, we construct the matrix B_{k+1} such that

$$B_{k+1}s_{k+1} = y_{k+1}, \text{ and} \tag{6.11}$$
$$B_{k+1}u = B_k u \quad \text{for all } u \quad \text{orthogonal to } s_{k+1}, \tag{6.12}$$

where

$$s_{k+1} := x_{k+1} - x_k, \text{ and}$$
$$y_{k+1} := f(x_{k+1}) - f(x_k).$$

Eqn. (6.11) is just a reformulation of the secant equation (6.10). Eqn. (6.12) guarantees that B_k and B_{k+1} act identically except for the one-dimensional subspace spanned by s_{k+1}.

Lemma 6.1. *The matrices B_k constructed according to the rank-1 update iteration:*

$$B_{k+1} = B_k + \frac{(y_{k+1} - B_k s_{k+1})\, s_{k+1}^T}{s_{k+1}^T s_{k+1}}, \tag{6.13}$$

fits the Eqns. (6.11) *and* (6.12).

Proof. We show that B_{k+1} fits into (6.11) and (6.12):

- First this is done by multiplying (6.13) by a u, such that $s_{k+1}^T u = 0$. Then, the second term on the r.h.s. vanishes, and therefore $B_{k+1}u = B_k u$ (6.12) remains.
- Furthermore, if we multiply (6.13) by s_{k+1}, we obtain $B_{k+1}s_{k+1} = B_k s_{k+1} + y_{k+1} - B_k s_{k+1} = y_{k+1}$, this is the secant equation (6.11).

□

The *Sherman-Morrison-Woodbury formula* [3] is now used to update the inverse of the Jacobian matrix directly (avoiding solving the linear system), yielding

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(s_{k+1} - B_k^{-1}y_{k+1})\,(s_{k+1}^T B_k^{-1})}{s_{k+1}^T B_k^{-1} y_{k+1}}. \tag{6.14}$$

If one already knows B_k^{-1}, then no extra inversion of a matrix is necessary to update B_k^{-1} according to (6.14). For initialization of B_1^{-1} the inverted Jacobian $J_f(x_0)$ can be calculated exactly or an estimate is used whose inverse is easy to compute. Quasi-Newton methods for unconstrained optimization often use a diagonal matrix approximation.
The method illustrated in Alg. 6 is referred to as *Broyden's method*.

Algorithm 6: Broyden's method

Data: function $f\colon \mathbb{R}^n \to \mathbb{R}^n$, initial guess x_0
Result: Approximate root x of $f(x)$
Initialization: Set $k \leftarrow 0, B_0^{-1} \leftarrow J_f(x_0)^{-1}$ or an approximation
while *Convergence criterion not satisfied* **do**

$$\begin{aligned} d_k &\leftarrow -B_k^{-1} f(x_k) \\ x_{k+1} &\leftarrow x_k + d_k \\ \text{Compute } B_{k+1}^{-1} &\text{ according to Eqn. (6.14)} \\ k &\leftarrow k+1 \end{aligned}$$

end
$x \leftarrow x_k$

Example 6.4. We use the same 'non-degenerate example' as before for testing Newton's method.
Let $f(x,y) = (\cos x - \sin y, xy - \pi/4)^T$ where $f(x,y) = 0$ has a non-degenerate solution at $x_* = (0.39876, 1.96956)^T$. We apply Broyden's method with $x_0 = (2,2)^T$ and finite difference approximation to the initial Jacobian and its exact inverse. Tab. 6.4 shows errors in $\|.\|_2$ and ratios of successive errors. As in the Newton case we observe Q-superlinear convergence but not more (e.g. Q-quadratic). □

[3] $(A + ab^T)^{-1} = A^{-1} - \frac{A^{-1}ab^T A^{-1}}{1+b^T A^{-1}a}$.

Table 6.4: Example 6.4 of root-finding Broyden's method.

k	e_k	e_k/e_{k-1}	e_k/e_{k-1}^2
0	1.6015	-	-
1	0.1206	0.0753	0.0470
2	0.0766	0.6347	5.2626
3	0.0450	0.5875	7.6750
4	0.0018	0.4026	8.9518
5	0.0013	0.0698	3.8544
6	0.0000	0.0062	4.8775
7	0.0000	0.1407	18056
8	0.0000	0.0420	38308
9	0.0000	0.0003	7307
10	0.0000	0.0002	10810144

Theorem 6.2 (local convergence, [10])**.** *Under the assumptions of Theorem 6.1, and if the starting guess x_0 is close enough to x_* and the starting Jacobian approximation B_0 are close enough to $J_f(x_0)$, Broyden's method (Alg. 6) is well-defined and convergence is Q-superlinear.* □

Remark 6.1. The storage requirements for the matrix B_k can be too expensive. To overcome this, note that B_{k+1}^{-1} is a sum of rank-1 components $a_j b_j^T$, $j = 1 \dots k$, so instead of building the full matrix, one can store the vectors $(a_j)_{j=1,\dots,k}$ and $(b_j)_{j=1,\dots,k}$. In this framework also *limited memory* variants can be considered: Only the last m vectors of the rank-1 components are stored and used for the updating scheme. □

Generally, for arbitrary initial values all inexact Newton methods (as also Newton's method itself) do not always converge. One possibility for obtaining a more globally convergent method is to use line search as described in the following section. Note that the local methods described so far still converge to local solutions (dependent on the initial guess) even with line search.

6.3 Basic line search concepts

The Newton method uses a local but exact linear model each iteration. As a consequence, Newton's method converges in one single step in the case of a linear function f. This is not true for inexact Newton methods, since they use an approximate linear model. It is therefore natural to consider a deviation from the paradigm of always using step-length 1. Hence, we will consider here a step length $t > 0$ for the update, i.e.

$$x_{k+1} = x_k + td. \tag{6.15}$$

Assume we have already calculated a direction $d \in \mathbb{R}^n$. We will try to find a step length t according to a (inexact) line search. More precisely, we search for

a step length that minimizes (resp. decreases in a specific way) a merit function w.r.t. the step length parameter t, for instance[4]

$$h(x_k + td) := \frac{1}{2}\|f(x_k + td)\|^2 = \frac{1}{2}\sum_{j=1}^{n} f_j^2(x_k + td). \tag{6.16}$$

Let us first discuss some details which may help to understand the forthcoming. We will denote the (univariate) function of t with

$$g(t) \equiv g(t; x_k, d) := h(x_k + td), \tag{6.17}$$

where x_k and d are assumed to be constant. Both, the multivariate function h and the univariate function g are differentiable.[5] Specifically, there holds for the derivative of g at $t = 0$

$$g'(0) = d^T \nabla h(x_k) = d^T \sum_{j=1}^{n} f_j(x_k) \nabla f_j(x_k) = d^T J_f(x_k)^T f(x_k), \tag{6.18}$$

where $J_f(x_k)$ is the Jacobian matrix of f at x_k. The quantity $g'(0) = d^T \nabla h(x_k)$ is also called the directional derivative of h along d at x_k. If $g'(0)$ is negative, then d is a direction of descent. More precisely, if we look at the first order Taylor expansion

$$h(x_k + td) = h(x_k) + td^T \nabla h(x_k) + \mathcal{O}(t^2), \tag{6.19}$$

we recognize that $h(x_k + td) - h(x_k)$ will be negative (descent) if $g'(0) = d^T \nabla h(x_k) < 0$ and t sufficiently small. Notably, it is not enough for convergence to only require a reduction in h, i.e. $h(x_k + td) < h(x_k)$. For instance, consider the convex function $\phi(x) = x^2 - 1$ with the unique minimizer $x_* = 0$ and $\phi(x_*) = -1$, and construct the iterates $x_k = (-1)^k \sqrt{1/k + 1}$. Obviously, $\phi(x_k) = 1/k$ is monotonically decreasing, but we have no convergence to the unique minimizer x_*.

Armijo's rule.
This situation does not occur if a *sufficient decrease condition* is imposed:

$$h(x_k + td) \leq h(x_k) + t\, c_1\, d^T \nabla h(x_k), \tag{6.20}$$

with $c_1 \in (0, 1)$. Note that condition (6.20) can also be written in terms of the univariate function g as

$$g(t) \leq g(0) + c_1 t g'(0). \tag{6.21}$$

Condition (6.20) is also referred to as *Armijo condition*. See Fig. 6.3 for an illustration of the Armijo condition. Note that because of $c_1 \in (0, 1)$ and $g'(0) <$

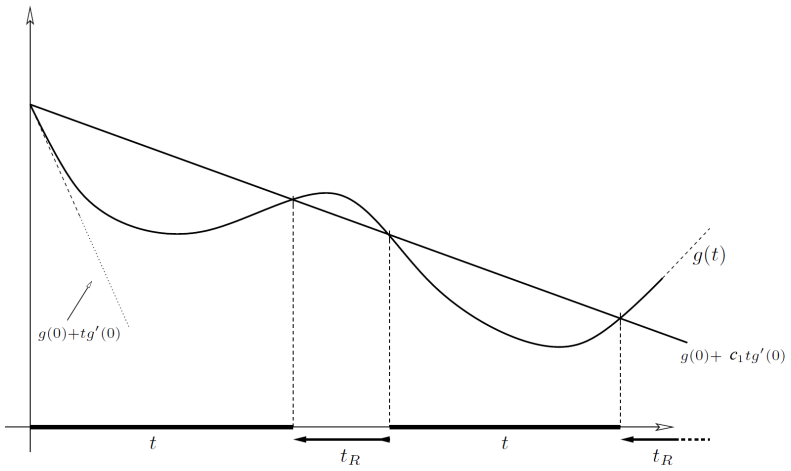

Figure 6.3: Armijo condition. Taken from [5]

0, the linear function in t defined by the r.h.s. of (6.20) lies above the function $g(t)$ for small positive values of t.

Algorithms which ensure condition (6.20) are often called *backtracking*, see Alg. 7. In practice, the parameter c_1 is chosen to be quite small, e.g. $c_1 = 10^{-4}$, [10].

Algorithm 7: Line search with Armijo condition.

Initialization: Choose some $t > 0$ and $0 < c_1 < 1$

while $g(t) > g(0) + c_1 t g'(0)$ **do**
  Decrease t, e.g. by fixed factor.
end
Define $t_* := t$

(Strong) Wolfe condition.
The Armijo condition never declares a step length as too small. In fact, for some methods (like Quasi-Newton) the Armijo condition is not enough for convergence. The so called *Wolfe condition* excludes values of t where the slope $g'(t)$ is too negative, while the *strong Wolfe condition* excludes those values of t where $|g'(t)|$ is too large (which is a stronger requirement). Note that such line search algorithms involve the computation of some values of $g'(t)$, which might be expensive. Therefore interpolation strategies are often incorporated, but stable and efficient methods are very elaborate.
More precisely, the Wolfe condition consists of the Armijo condition and a *cur-*

[4]Note that the quadratic norm is differentiable.
[5]Note that we assume f to be differentiable and note that h in (6.16) is differentiable w.r.t. any variable.

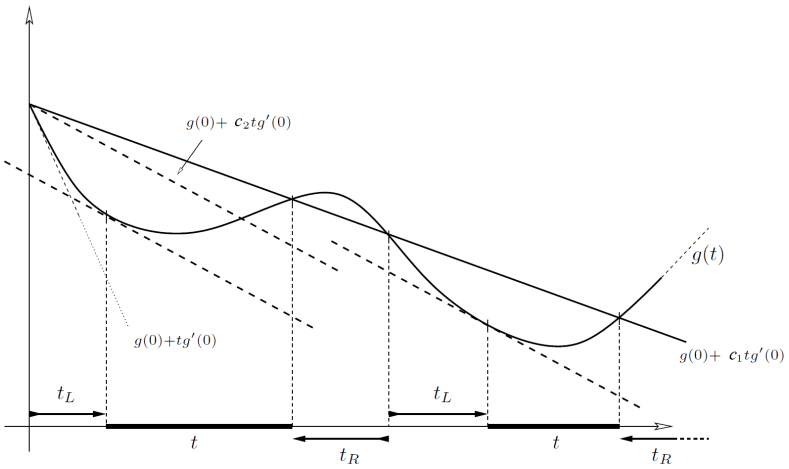

Figure 6.4: Wolfe condition. In addition to the Armijo condition, the slope $g'(t)$ is required to be less negative than the desired slope $c_2\, g'(0)$ (which lies between $g'(0)$ and $c_1 g'(0)$). Figure taken from [5].

vature condition

$$g'(t) \geq c_2\, g'(0), \quad c_2 \in (c_1, 1). \tag{6.22}$$

See Fig. 6.4 for an illustration of the Wolfe condition. See Alg. 8 for an algorithmic description.

In practice, the parameter c_2 is chosen loosely (for Newton or Quasi-Newton), e.g. $c_2 = 0.9$ or quite tight (for nonlinear conjugate gradient methods), e.g. $c_2 = 0.1$, depending on the method [10].
The *strong Wolfe condition* consists of the Armijo condition and the stronger *curvature condition*

$$|g'(t)| \leq c_2\, |g'(0)|, \quad c_2 \in (c_1, 1). \tag{6.23}$$

Existence for Wolfe step.
Assume d being a descent direction at x_k ($d^T \nabla h(x_k) < 0$). Then, one can prove that if the function which is minimized (here h from (6.19)) is continuously differentiable and bounded (from below) along the line $\{x_k + td | t > 0\}$, then there exist step lengths (intervals) satisfying the Wolfe condition and the strong Wolfe condition [10].

Convergence of line-search Newton.
Under relatively strong regularity assumptions ($J_f(x)$ Lipschitz continuous) the Newton method supplemented with a line search that satisfies the sufficient decrease condition and the Wolfe condition is guaranteed to converge globally (independent of starting guess) to a stationary point of the merit function h. If the Jacobians at the iterates are bounded then the line search Newton method

Algorithm 8: Wolfe line search.

Initialization: set $t_L = 0$ and $t_R = +\infty$ choose some initial $t > 0$;
declare t unacceptable, fix $0 < c_1 \leq c_2 < 1$;

while t *is unacceptable* **do**
  if $g(t) > g(0) + c_1 t g'(0)$ **then**
    set $t_R \leftarrow t$;
    choose new $t \in (t_L, t_R)$
  else if $g'(t) < c_2 g'(0)$ **then**
    set $t_L \leftarrow t$;
    choose new $t \in (t_L, t_R)$
  else
    declare t acceptable
  end
end
define $t_* := t$;

converges to a solution of the equation $f(x) = 0$. See e.g. [10] for further information on this topic.

Goldstein and Price.
The (strong) Wolfe condition (see Alg. 8) requires the evaluation of $g'(t)$ which might be very expensive. A possible way out might be to replace the derivative by a secant, like $(g(t) - g(0))/t$. The rule of *Goldstein and Price* (see Fig. 6.5) defines a step length acceptable if

$$c_2 g'(0) \leq \frac{g(t) - g(0)}{t} \leq c_1 g'(0). \tag{6.24}$$

Note that this includes the Armijo rule (upper bound). The Goldstein and Price algorithm is summarized in Alg. 9.

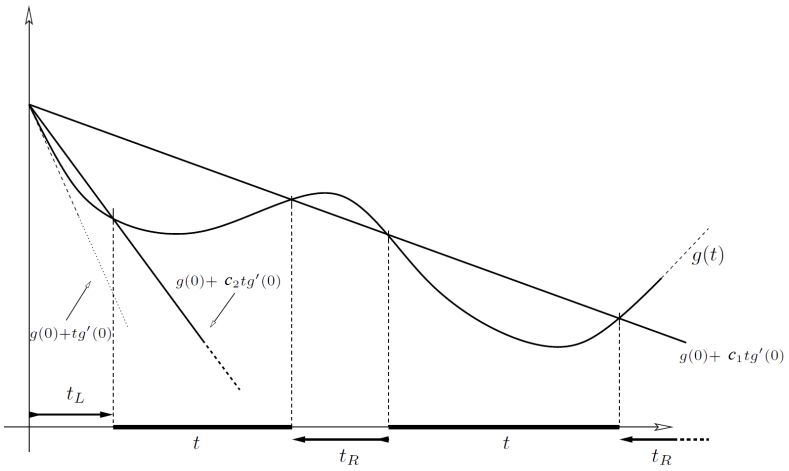

Figure 6.5: Goldstein and Price line search. In addition to Armijo's rule, the slope of the secant $s(\tau) = g(0) + \tau(g(t) - g(0))/t$ is required to be less negative than the desired slope $c_2\, g'(0)$ (which lies between $g'(0)$ and $c_1 g'(0)$). Figure taken from [5].

Algorithm 9: Line search according to Goldstein and Price.

Initialization: Set $t_L = 0$ and $t_R = +\infty$ choose some initial $t > 0$;
declare t unacceptable, fix $0 < c_1 < c_2 < 1$;
while t *is unacceptable* **do**
  if $g(t) > g(0) + c_1 t g'(0)$ **then**
    set $t_R \leftarrow t$;
    choose new $t \in (t_L, t_R)$
  else if $g(t) < g(0) + c_2 t g'(0)$ **then**
    set $t_L \leftarrow t$;
    choose new $t \in (t_L, t_R)$
  else
    declare t acceptable
  end
end
define $t_* := t$

Appendix A
Linear Algebra

Throughout this document $\mathbb{K}$ will stand for either the set of real numbers $\mathbb{R}$ or the set of complex numbers $\mathbb{C}$. Whenever a statement is true for both $\mathbb{R}$ and $\mathbb{C}$ we use the symbol $\mathbb{K}$, and we restrict the use of $\mathbb{R}$ and $\mathbb{C}$ to those situations, where the actual choice makes a difference.

A.1 Linear dependence

Let $n \in \mathbb{N} = \{1, 2, \ldots\}$ be a natural number. We follow the convention that elements of the vector space $\mathbb{K}^n$ are identified with column vectors. That is, $x \in \mathbb{K}^n$, if

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad x_1, \ldots, x_n \in \mathbb{K}.$$

Let $x_1, \ldots, x_k \in \mathbb{K}^n$ be a set of vectors. Every sum of the form

$$\alpha_1 x_1 + \cdots + \alpha_k x_k$$

with $\alpha_i \in \mathbb{K}$ is called a *linear combination* of the vectors x_i. The set of all possible linear combinations is called *span* of $x_1, \ldots, x_k$, in symbols

$$\operatorname{span}\{x_1, \ldots, x_k\} = \{\alpha_1 x_1 + \cdots + \alpha_k x_k : \alpha_i \in \mathbb{K}\}.$$

A *subspace* of $\mathbb{K}^n$ is a subset $S \subset \mathbb{K}^n$ with the additional property that every linear combination of vectors in S lies again in S. That is, if for all $x, y \in S$ and scalars $\alpha, \beta \in \mathbb{K}$ we have $\alpha x + \beta y \in S$, then S is a subspace. The span of a set of vectors in $\mathbb{K}^n$ is always a subspace of $\mathbb{K}^n$. A set of vectors is called *linearly dependent*, if there is a nontrivial linear combination (not all $\alpha_i = 0$) that equals the zero vector. If there is no such linear combination, the set is called *linearly independent*. Let S be a subspace of $\mathbb{K}^n$. A linearly independent set of vectors whose span equals S is called *basis of* S. The *dimension* of a (sub)space is the number of elements of any of its bases. An exemplary statement that contains most notions introduced above is this: If the vectors $x_1, \ldots, x_k$ are linearly independent, then they form a basis of their span, which in this case is k-dimensional.

A.2 Matrix products

Let $A = (a_{ij}) \in \mathbb{K}^{m\times n}$ be a matrix with m rows and n columns. We follow the usual convention of denoting by a_{ij} the entry in row i and column j. For every $x \in \mathbb{K}^n$ the matrix-vector product $Ax = b \in \mathbb{K}^m$ is defined by

$$b_i = \sum_{j=1}^{n} a_{ij}x_j, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}.$$

In this way, every matrix defines a function from $\mathbb{K}^n$ to $\mathbb{K}^m$

$$A : \mathbb{K}^n \to \mathbb{K}^m, x \mapsto Ax.$$

Every such function is *linear*, meaning that $A(\alpha x + \beta y) = \alpha Ax + \beta Ay$ for all $x, y \in \mathbb{K}^n$ and all $\alpha, \beta \in \mathbb{K}$. Conversely, it can be shown that every linear function between $\mathbb{K}^n$ and $\mathbb{K}^m$ can be represented by a matrix $A \in \mathbb{K}^{m\times n}$. Denote the j-th column of A by a_j. Then we can rewrite the matrix-vector product as $Ax = \sum_{j=1}^{n} x_j a_j$. Thus, Ax is a linear combination of the columns of A.

Let $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times k}$. The matrix-matrix product $AB = C \in \mathbb{K}^{m\times k}$ is defined by

$$c_{ij} = \sum_{\ell=1}^{n} a_{i\ell}b_{\ell j}, \quad 1 \le i \le m, \quad 1 \le j \le k.$$

Note that the product AB can only be formed, if the number of columns of A equals the number of rows of B. As before denote the j-th columns of B and C by b_j and c_j respectively. Then, c_j equals the matrix vector product Ab_j. In other words the j-th column of C is a linear combination of the columns of A where the coefficients are given by the entries of the j-th column of B. Matrix-matrix multiplication is *associative*, that is, $A(BC) = (AB)C$ for all matrices $A \in \mathbb{K}^{m\times n}$, $B \in \mathbb{K}^{n\times k}$ and $C \in \mathbb{K}^{k\times \ell}$. It is also *distributive* over addition

$$A(B + C) = AB + AC, \qquad (A + B)C = AC + BC.$$

Matrix-matrix multiplication is *not commutative*, however, which means that $AB = BA$ does not hold for all matrices $A, B \in \mathbb{K}^{n\times n}$.

A.3 Invertibility

The *range* of a matrix $A \in \mathbb{K}^{m\times n}$ is simply its range when regarded as a function from $\mathbb{K}^n$ to $\mathbb{K}^m$

$$\operatorname{ran} A = \{Ax : x \in \mathbb{K}^n\}.$$

The interpretation of the matrix-vector product as a linear combination implies that $\operatorname{ran} A$ is equal to the span of the columns of A. This is the reason why $\operatorname{ran} A$ is sometimes called *column space* of A. The *row space* is defined analogously. It

can be shown that the column and row spaces of a matrix always have the same dimension. This number is called *rank* of the matrix and denoted by $\operatorname{rank} A$. If $\operatorname{rank} A$ is maximal, that is, $\operatorname{rank} A = \min(m, n)$, then A is said to have *full rank*. The *kernel* or *nullspace* of a matrix is the set of all vectors that are mapped to zero by A

$$\ker A = \{x \in \mathbb{K}^n : Ax = 0\}.$$

The *identity* or *unit matrix* $I \in \mathbb{K}^{n\times n}$ is the matrix which has ones along its main diagonal and zeros elsewhere. It satisfies $IA = A$ for all matrices A with n rows and $BI = B$ for all matrices B with n columns. Using the *Kronecker delta*

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

the identity matrix can be defined succinctly as $I = (\delta_{ij})$. Let $A \in \mathbb{K}^{n\times n}$ be a square matrix. If there is another matrix B satisfying $AB = BA = I$, then A is called *invertible* or *regular*. In this case, the matrix B is unique, it is called the *inverse* of A and it is denoted by A^{-1}. The inverse of a product of two regular matrices is the reversed product of the inverse matrices: $(AB)^{-1} = B^{-1}A^{-1}$. A matrix which does not have an inverse is called *singular*.

Theorem A.1 (Characterization of regular matrices)**.** *Let $A \in \mathbb{K}^{n\times n}$. The following statements are equivalent.*

1. *A is regular.*
2. *A is injective.*
3. *For every $b \in \mathbb{K}^n$ the linear system $Ax = b$ has exactly one solution.*
4. $\operatorname{ran} A = \mathbb{K}^n$.
5. $\operatorname{rank} A = n$.
6. $\ker A = \{0\}$.
7. $\det A \neq 0$.
8. *0 is not an eigenvalue of A.*

The determinant $\det A$ and eigenvalues of a matrix are introduced below.

A.4 Orthogonality

The *transpose* $A^\top \in \mathbb{K}^{n\times m}$ of a matrix $A \in \mathbb{K}^{m\times n}$ is defined by $a_{ij}^\top = a_{ji}$. The *adjoint* or *conjugate transpose* A^* of A is defined by $A^* = \bar{A}^\top$, where the bar means complex conjugation of all entries. If $A^* = A$, then A is called *self-adjoint*. In the complex case ($\mathbb{K} = \mathbb{C}$) A is also called *Hermitian*. A real self-adjoint matrix, that is one satisfying $A^\top = A$, is also called *symmetric*.

The adjoint of a product of two matrices is the reversed product of the adjoint matrices: $(AB)^* = B^*A^*$.

Using the $*$-notation we define the *inner product* between two vectors $x, y \in \mathbb{K}^n$ as x^*y. (Note that every vector can be regarded as a single-column matrix.) For real vectors the inner product is symmetric: $x^*y = y^*x$. For complex ones it is not. The *Euclidean norm* on $\mathbb{K}^n$ can be defined via the inner product as $\|x\|_2 = \sqrt{x^*x}$.[1] The inner product also allows one to measure the angle α between vectors

$$\cos\alpha = \frac{x^*y}{\|x\|_2\|y\|_2}.$$

Two vectors $x, y \in \mathbb{K}^n$ are *orthogonal*, if their inner product equals zero. A set of more than two nonzero vectors is called orthogonal, if they are all mutually orthogonal. A set of vectors $q_1, \ldots, q_\ell$ is called *orthonormal*, if it is orthogonal and all vectors have norm 1. This can be stated succinctly as $q_i^*q_j = \delta_{ij}$ for all i, j. Sets of nonzero orthogonal vectors are linearly independent. Thus, every orthogonal set consisting of n vectors in $\mathbb{K}^n$ is a basis of $\mathbb{K}^n$. The simplest orthonormal basis of $\mathbb{K}^n$ is the *canonical basis* $\{e_1, \ldots, e_n\}$, given by $e_{i,j} = \delta_{ij}$.

Orthonormal vectors are convenient to work with. One reason is that they allow us to easily decompose any given vector into orthogonal components. To demonstrate this let $q_1, \ldots, q_k \in \mathbb{K}^n$ be an orthonormal set and $x \in \mathbb{K}^n$ an arbitrary vector. Then the vector

$$r = x - (q_1^*x)q_1 - \cdots - (q_k^*x)q_k \tag{A.1}$$

is orthogonal to all q_i. Multiply both sides with q_i^* from the left to see this! Thus we have found a way to write x as a sum of $k+1$ mutually orthogonal vectors. This decomposition suggests a procedure for turning a linearly independent set of vectors into an orthonormal set:

Let $x_1, \ldots, x_\ell \in \mathbb{K}^n$ be linearly independent. We construct an orthonormal set $q_1, \ldots, q_\ell$ inductively. Set $q_1 = x_1/\|x_1\|_2$. Then q_1 has norm one. To find the second vector we set

$$p_2 = x_2 - (q_1^*x_2)q_1.$$

By (A.1) this vector is orthogonal to q_1. After normalizing $q_2 = p_2/\|p_2\|_2$, the set $\{q_1, q_2\}$ is orthonormal. Now, suppose you have already found k orthonormal vectors, where $k < \ell$. Then set

$$p_{k+1} = x_{k+1} - (q_1^*x_{k+1})q_1 - \cdots - (q_k^*x_{k+1})q_k \tag{A.2}$$

and normalize again to get q_{k+1}. After ℓ steps we have found an orthonormal set with the property

$$\operatorname{span}\{q_1, \ldots, q_k\} = \operatorname{span}\{x_1, \ldots, x_k\}$$

for all k between 1 and ℓ. This is the *Gram-Schmidt process.*

[1] A more general definition of norms as well as an explanation of the subscript 2 in $\|\cdot\|_2$ will be given in Sect. A.8 below.

A square matrix $Q \in \mathbb{C}^{n\times n}$ is called *unitary*, if $Q^*Q = QQ^* = I$. Put differently, the inverse and the adjoint of Q coincide. A square matrix $Q \in \mathbb{R}^{n\times n}$ with the analogous property $Q^{-1} = Q^\top$ is called *orthogonal*. It is easy to see that the columns of Q form an orthonormal basis of $\mathbb{K}^n$: Denote the j-th column of Q by q_j. Then the (i, j) entry of Q^*Q equals $q_i^*q_j$. On the other hand the (i, j) entry of I equals δ_{ij}. Since $Q^*Q = I$, the columns of Q are an orthonormal basis of $\mathbb{K}^n$. An analogous argument for QQ^* instead of Q^*Q shows that the rows form an orthonormal basis as well.

Orthogonal and unitary matrices preserve the geometrical structure of the underlying space. Applying such a matrix to a pair of vectors does not change their inner product:

$$(Qy)^*Qx = y^*Q^*Qx = y^*x. \tag{A.3}$$

Consequently it does not change norms or angles either. Orthogonal matrices have a nice geometrical interpretation. They are all rotations, reflections or combinations of thereof. For example, the orthogonal matrix

$$\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

rotates every vector in $\mathbb{R}^2$ by an angle α around the origin. On the other hand, let $x \in \mathbb{K}^n$ be an arbitrary unit vector ($\|x\|_2 = 1$). Then the *reflection matrix* $I - 2xx^*$, which reflects every vector across the hyperplane[2] orthogonal to x, is an orthogonal/unitary matrix as well. Note that the expression xx^* is an *outer product*. For two general vectors $x \in \mathbb{K}^m$ and $y \in \mathbb{K}^n$ the outer product xy^* is the $m \times n$ matrix given by

$$xy^* = \begin{bmatrix} x_1\bar{y}_1 & \cdots & x_1\bar{y}_n \\ \vdots & \ddots & \vdots \\ x_m\bar{y}_1 & \cdots & x_m\bar{y}_n \end{bmatrix}.$$

The outer product can also be regarded as the matrix-matrix product of the $m \times 1$ matrix x and the $1 \times n$ matrix y^*.

A.5 Projections

A *projection (matrix)* is a square matrix that satisfies $P^2x = Px$ for all x. Here, P^2 stands for the matrix-matrix product PP. Applying a projection matrix more than once to a vector does not change the result anymore. For example, identity matrices with an arbitrary number of ones deleted are projection matrices. If P is a projection, then

$$(I - P)^2x = (I - P)(x - Px) = x - Px - Px + Px = (I - P)x$$

[2] A hyperplane is an $(n - 1)$-dimensional subspace of $\mathbb{K}^n$, e.g. a line through the origin in $\mathbb{R}^2$ and a plane through the origin in $\mathbb{R}^3$.

and therefore $I - P$ is a projection as well, the *complementary projection* to P. The complementary projection to $I-P$ is again P. Every projection decomposes $\mathbb{K}^n$ into the two subspaces $\operatorname{ran} P$ and $\ker P$ in the sense that

$$\operatorname{ran} P + \ker P = \mathbb{K}^n, \quad \text{while} \quad \operatorname{ran} P \cap \ker P = \{0\}.$$

The first statement above means that for every $x \in \mathbb{K}^n$, there is a $x_1 \in \operatorname{ran} P$ and a $x_2 \in \ker P$ such that $x = x_1 + x_2$. Also note that the kernel of a projection equals the range of its complementary projection: $\ker P = \operatorname{ran}(I - P)$.

A projection is said to be *orthogonal,* if $\operatorname{ran} P$ is orthogonal to $\ker P$. This means that $x^*y = 0$ for all $x \in \ker P$ and $y \in \operatorname{ran} P$. An alternative characterization is this: A projection is orthogonal if and only if it is self-adjoint. The complementary projection to an orthogonal projection is again orthogonal, since $(I - P)^* = I^* - P^* = I - P$.

Every set of orthonormal vectors $q_1, \ldots, q_k \in \mathbb{K}^n$ can be used to construct an orthogonal projection. Let $Q \in \mathbb{K}^{n\times k}$ be the matrix whose j-th column is q_j and define $P = QQ^*$. This matrix is an orthogonal projection, because

$$\begin{aligned} P^2 &= QQ^*QQ^* = QIQ^* = P, \quad \text{and} \\ P^* &= (QQ^*)^* = Q^{**}Q^* = P. \end{aligned}$$

Notice that the j-th entry of the vector Q^*x is q_j^*x and therefore, by the linear combination interpretation of the matrix-vector product, we have

$$Px = QQ^*x = \sum_{j=1}^{k} (q_j^*x)q_j.$$

From this expression it can be deduced that the range of P is the span of $\{q_1, \ldots, q_k\}$.

The previous paragraph sheds new light on the orthogonal decomposition (A.1), which we can now rewrite as $r = (I - P)x$. The fact that r is orthogonal to all q_j can be deduced from what we have shown above. We have

$$r \in \operatorname{ran}(I - P) = \ker P \perp \operatorname{ran} P = \operatorname{span}\{q_1, \ldots, q_k\},$$

where the symbol $\perp$ means "orthogonal to."

Now we can also reformulate the Gram-Schmidt process using our new terminology. Suppose we have found k orthonormal vectors $q_1, \ldots, q_k$. Arrange them as columns into a matrix $Q_k \in \mathbb{K}^{n\times k}$ and define the projection $P_k = Q_kQ_k^*$. Then

$$p_{k+1} = x_{k+1} - Q_kQ_k^*x_{k+1} = (I - P_k)x_{k+1},$$

and the normalization of p_{k+1} is as before.

A.6 Determinant

The *determinant*, $\det A \in \mathbb{K}$, is a scalar associated to every square matrix A. It can be used to determine whether a matrix is regular or singular, recall Theorem A.1. It is also useful for finding the eigenvalues of a matrix. Geometrically, the determinant of a 2×2 matrix is related to the area of the parallelogram spanned its columns or rows. More precisely, we have

$$|\det A| = \text{area}\,(P),$$

where P is the parallelogram with vertices 0, a_1, a_2 and $a_1 + a_2$. Similarly, if A is 3×3, then

$$|\det A| = \text{vol}\,(P),$$

where P is the parallelepiped with vertices 0, a_1, a_2, a_3, $a_1 + a_2$, $a_1 + a_3$, $a_2 + a_3$ and $a_1 + a_2 + a_3$. This geometric interpretation is the reason why the determinant appears in multidimensional integration-by-substitution formulas.

The determinant of a 2×2 matrix is given by

$$\det A = a_{11}a_{22} - a_{12}a_{21}.$$

For $n \times n$ matrices we can use *Laplace expansion* to express the determinant recursively. Fix an index $i \in \{1, \dots, n\}$. Then the Laplace expansion of $\det A$ along row i reads

$$\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det A_{ij},$$

where A_{ij} is the submatrix of A obtained by removing row i and column j. There is an analogous formula for Laplace expansion along column j

$$\det A = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \det A_{ij}.$$

Note that A_{ij} is $(n-1)\times(n-1)$ so that Laplace expansion reduces the calculation of one $n \times n$ determinant to n determinants of size $(n-1) \times (n-1)$. Therefore, practically speaking it makes sense to expand along that row or column which has the most zeros.

While it can be tedious to calculate the determinant for a general matrix, there are certain special matrices and rules of calculation which allow for relatively simple formulas.

1. If A is triangular, then its determinant is the product of its diagonal entries

$$\det A = a_{11}a_{22}\cdots a_{nn}. \tag{A.4}$$

   Recall that a triangular matrix has only zero entries on one side of its main diagonal. There are two possibilities: An *upper triangular matrix* satisfies $a_{ij} = 0$ whenever $i > j$. A *lower triangular matrix* has $a_{ij} = 0$ whenever $i < j$.

2. Taking the transpose does not alter the determinant

$$\det A^\top = \det A.$$

   This implies that we can replace the word "row" by "column" in items 3 and 4 below. Moreover,

$$\det \bar{A} = \overline{\det A}.$$

   Combining this with the previous property yields the following formula for the adjoint

$$\det(A^*) = \overline{\det A}.$$

3. Adding a multiple of one row to another does not change the determinant. This means we can use Gaussian elimination to bring a matrix into triangular form (if possible), and then use equation (A.4).

4. Interchanging two rows changes the sign of the determinant. Therefore we can also use Gaussian elimination with pivoting as long as we take care of the sign changes.

5. The determinant of a matrix product is the product of determinants

$$\det(AB) = \det A \det B.$$

   This implies that $\det(A^{-1}) = 1/\det A$. It also implies that orthogonal/unitary matrices satisfy $|\det Q| = 1$.

6. The determinant is equal to the product of eigenvalues of the matrix

$$\det A = \lambda_1 \cdots \lambda_n.$$

   Note that if an eigenvalue has multiplicity m as a zero of the characteristic polynomial, then it appears m times in the formula above. See Sect. A.7.

A.7 Eigenvalues

A scalar $\lambda \in \mathbb{C}$ is an *eigenvalue* of $B \in \mathbb{K}^{n\times n}$, if there is a nonzero vector x such that

$$Bx = \lambda x. \tag{A.5}$$

In this case x is an *eigenvector* of B. The set of all eigenvalues of B is the *spectrum* of B, and the *spectral radius* is defined as

$$\rho(B) = \max\{|\lambda| : \lambda \text{ eigenvalue of } B\}.$$

Geometrically speaking, eigenvectors indicate directions along which a matrix only stretches, shortens (or reflects if $\lambda < 0$). The corresponding eigenvalues give the amount of stretching or shortening.

Equation (A.5) can be rewritten as $(\lambda I - B)x = 0$. From items 6 and 7 in Theorem A.1 it now follows that

$$\det(\lambda I - B) = 0.$$

The determinant on the left is a polynomial of degree n in λ, the *characteristic polynomial* of B. Thus, finding the eigenvalues of B amounts to finding the zeros of its characteristic polynomial. By the fundamental theorem of algebra we can rewrite it as

$$\det(\lambda I - B) = (\lambda - \lambda_1)(\lambda - \lambda_2)\cdots(\lambda - \lambda_n),$$

where the zeros λ_i need not have distinct values and can be complex even if B is real. Note that, if B is real, then the eigenvectors corresponding to complex eigenvalues must be complex too.

The following theorem is one of the most important in linear algebra.

Theorem A.2 (Spectral theorem)**.** *If $B \in \mathbb{K}^{n\times n}$ is self-adjoint, then all its eigenvalues are real and there is an orthonormal basis of $\mathbb{K}^n$ consisting of eigenvectors of B. Put differently, there is an orthogonal/unitary matrix $Q \in \mathbb{K}^{n\times n}$ such that*

$$B = Q\Lambda Q^*,$$

where $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$.

The square roots of the eigenvalues of A^*A are called *singular values* of $A \in \mathbb{K}^{m\times n}$. They are typically denoted $\sigma_1, \ldots, \sigma_n$. Singular values are always nonnegative real numbers. This can be seen by multiplying the equation $A^*Ax = \lambda x$ from the left with x^*. This leads to

$$\|Ax\|_2^2 = \lambda \|x\|_2^2,$$

which implies $\lambda \geq 0$, and consequently $\sqrt{\lambda} \geq 0$. The largest (smallest) singular value of a matrix is denoted by $\sigma_{\max}$ ($\sigma_{\min}$).

A.8 Norms

A *vector norm* is a function

$$\|\cdot\| : \mathbb{K}^n \to [0, +\infty)$$

that assigns a length to each vector. It must satisfy the following three conditions. First, the zero vector is the only vector with length 0. Second, if a vector is multiplied by a scalar, then its length should scale accordingly. Third, the length of a vector must never exceed the combined lengths of its constituents. More precisely, for all $\alpha \in \mathbb{K}$ and $x, y \in \mathbb{K}^n$

1. $\|x\| = 0$ if and only if $x = 0$,

2. $\|\alpha x\| = |\alpha|\|x\|$,
3. $\|x + y\| \leq \|x\| + \|y\|$ (*triangle inequality*).

An important family of norms are the *p-norms*

$$\|x\|_p = (|x_1|^p + \cdots + |x_n|^p)^{\frac{1}{p}}, \quad p \in [1, +\infty),$$

and the *maximum norm*

$$\|x\|_\infty = \max_{1 \leq i \leq n} |x_i|.$$

The Euclidean norm, which we introduced above using the inner product, corresponds to the value $p = 2$.

Two norms $\|\cdot\|$ and $\|\cdot\|'$ on $\mathbb{K}^n$ are called *equivalent*, if there are positive constants C_1 and C_2 such that the inequality

$$C_1\|x\| \leq \|x\|' \leq C_2\|x\| \tag{A.6}$$

holds for all $x \in \mathbb{K}^n$. It can be shown that *all* vector norms are equivalent. (The same holds true for matrix norms, see below.) For example, we have

$$\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n}\|x\|_\infty$$

for all $x \in \mathbb{R}^n$. Because of this equivalence certain statements involving norms hold true independently of the choice of norm.

There are two ways of looking at $m \times n$ matrices. First, we can view them as elements of an mn-dimensional vector space. Second, we can interpret them as linear maps between $\mathbb{K}^n$ and $\mathbb{K}^m$. The first viewpoint leads to a definition analogous to the one for vectors: A *matrix norm* is a function $\|\cdot\| : \mathbb{K}^{m\times n} \to [0, +\infty)$ that satisfies, for all $\alpha \in \mathbb{K}$ and $A, B \in \mathbb{K}^{m\times n}$, the following conditions

1. $\|A\| = 0$ if and only if $A = 0$,
2. $\|\alpha A\| = |\alpha|\|A\|$,
3. $\|A + B\| \leq \|A\| + \|B\|$.

The second viewpoint leads to a definition of matrix norm that depends on the respective norms on $\mathbb{K}^n$ and $\mathbb{K}^m$: Let $\|\cdot\|_{(n)}$ be a norm on $\mathbb{K}^n$ and $\|\cdot\|_{(m)}$ a norm on $\mathbb{K}^m$. Then the *matrix norm induced by* $\|\cdot\|_{(n)}$ *and* $\|\cdot\|_{(m)}$ is defined as

$$\|A\|_{(n,m)} = \max_{x \neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}}. \tag{A.7}$$

Intuitively, the ratio $\|Ax\|_{(m)}/\|x\|_{(n)}$ measures the amount by which A has stretched or shortened the vector x. Thus, the number $\|A\|_{(n,m)}$ measures the maximal change of length that the matrix A induces on $\mathbb{K}^n$.

Let us collect some properties of induced matrix norms. First, every induced matrix norm satisfies the three conditions of a matrix norm above. Next, we can deduce directly from the definition that

$$\|Ax\|_{(m)} \leq \|A\|_{(n,m)}\|x\|_{(n)}$$

for all x. Consequently, we have for matrices $A \in \mathbb{K}^{m\times n}$ and $B \in \mathbb{K}^{n\times k}$

$$\|ABx\|_{(m)} \le \|A\|_{(n,m)}\|Bx\|_{(n)} \le \|A\|_{(n,m)}\|B\|_{(k,n)}\|x\|_{(k)}.$$

Dividing by $\|x\|_{(k)}$ and taking the maximum over all $x \neq 0$ shows the so-called *submultiplicativity* of induced matrix norms:

$$\|AB\|_{(k,m)} \le \|A\|_{(n,m)}\|B\|_{(k,n)}.$$

It is sometimes useful to realise that in (A.7) it is actually enough to take the maximum only over those $x \in \mathbb{K}^n$ with $\|x\| = 1$

$$\|A\|_{(n,m)} = \max_{x\neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} = \max_{x\neq 0} \|A(x/\|x\|_{(n)})\|_{(m)} = \max_{\|y\|_{(n)}=1} \|Ay\|_{(m)}. \quad \text{(A.8)}$$

Suppose that A is invertible. Then the induced norm of A^{-1} can be written as

$$\|A^{-1}\| = \max_{x\neq 0} \frac{\|A^{-1}x\|}{\|x\|} = \max_{y\neq 0} \frac{\|y\|}{\|Ay\|} = \left(\min_{y\neq 0} \frac{\|Ay\|}{\|y\|}\right)^{-1} = \left(\min_{\|y\|=1} \|Ay\|\right)^{-1}. \quad \text{(A.9)}$$

The third equality above follows from the fact that the minimum of the reciprocal of a function equals the reciprocal of the maximum of the same function. More precisely,

$$\left(\max_{x\in X} f(x)\right)^{-1} = \min_{x\in X} \frac{1}{f(x)},$$

for every strictly positive function $f : X \to (0,\infty)$ attaining a maximum on X, where X is an arbitrary set.

The matrix norm induced by the p-norm on both $\mathbb{K}^n$ and $\mathbb{K}^m$ is denoted in the same way as the vector norm, i.e. by a subscript p:

$$\|A\|_p = \max_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

Example A.1 ($p = 1$)**.** The matrix norm induced by the 1-norm is given by

$$\|A\|_1 = \max_{1\le j\le n} \|a_j\|_1, \quad \text{(A.10)}$$

where a_j denotes the j-th column of A. This can be shown as follows. We have

$$\|Ax\|_1 = \left\|\sum_{j=1}^n x_j a_j\right\|_1 \le \sum_{j=1}^n |x_j|\|a_j\|_1 \le \sum_{j=1}^n |x_j| \max_{1\le k\le n} \|a_k\|_1 = \|x\|_1 \max_{1\le k\le n} \|a_k\|_1$$

for all $x \in \mathbb{K}^n$. Dividing by $\|x\|_1$ gives $\|A\|_1 \le \max_j \|a_j\|_1$. For the reverse inequality, we may assume without limitation of generality that the ℓ-th column has the largest 1-norm, that is, $\|a_\ell\|_1 = \max_j \|a_j\|_1$. We get

$$\|A\|_1 \ge \|Ae_\ell\|_1 = \|a_\ell\|_1 = \max_{1\le j\le n} \|a_j\|.$$

Example A.2 ($p = \infty$)**.** The matrix norm induced by the ∞-norm is given by

$$\|A\|_\infty = \max_{1\le i\le m} \|r_i\|_1,$$

where r_i denotes the i-th row of A. We proceed similarly to the case $p = 1$. For all $x \in \mathbb{K}^n$ we have

$$\|Ax\|_\infty = \max_{1\le i\le m} |r_i^* x| \le \max_{1\le i\le m} \|r_i\|_1 \|x\|_\infty,$$

yielding $\|A\|_\infty \le \max_i \|r_i\|_1$. For the reverse equality we assume that $\max_i \|r_i\|_1 = \|r_\ell\|_1$ and define the vector y by $y_j = \operatorname{sgn}(a_{\ell j}) = \operatorname{sgn}(r_{\ell,j})$. Then

$$\|A\|_\infty \ge \|Ay\|_\infty = \max_{1\le i\le m} |r_i^* y| \ge |r_\ell^* y| = \|r_\ell\|_1 = \max_i \|r_i\|_1.$$

Example A.3 ($p = 2$)**.** The matrix norm induced by the Euclidean norm equals the largest singular value of A, in symbols

$$\|A\|_2 = \sqrt{\rho(A^*A)} = \sigma_{\max}. \tag{A.11}$$

Hence its name *spectral norm*. To show (A.11) we write

$$\|A\|_2^2 = \max_{\|x\|_2=1} \|Ax\|_2^2 = \max_{\|x\|_2=1} x^* A^* A x.$$

Now we use the fact that A^*A is self-adjoint and therefore can be diagonalized by an orthogonal/unitary Q, recall Theorem A.2. In addition every x with length one can be written as $x = Qy$, where y also has length one, so that we can take the maximum over all y instead.

$$\|A\|_2^2 = \max_{\|y\|_2=1} y^* Q^* A^* A Q y = \max_{\|y\|_2=1} y^* \Lambda y.$$

Note that the diagonal matrix Λ now contains the squared singular values of A: $\Lambda = \operatorname{diag}(\sigma_1^2, \dots, \sigma_n^2)$. Therefore

$$\|A\|_2^2 = \max_{\|y\|_2=1} \left(\sigma_1^2 |y_1|^2 + \cdots + \sigma_n^2 |y_n|^2\right).$$

The sum on the right is maximized if we set the y_i which multiplied with the largest singular value equal to one, and all others to zero. Consequently,

$$\|A\|_2^2 = \sigma_{\max}^2,$$

which is equivalent to (A.11). An analogous calculation, taking into account equation (A.9), shows that

$$\|A^{-1}\|_2 = \frac{1}{\sigma_{\min}}.$$

One remarkable property of the spectral norm is that it is invariant under orthogonal or unitary transformations. Let U and V be two such matrices. Then

$$\begin{aligned}\|UAV\|_2^2 &= \max_{\|x\|_2=1} x^*V^*A^*U^*UAVx = \max_{\|x\|_2=1} \|AVx\|_2^2 \\ &= \max_{\|Vx\|_2=1} \|AVx\|_2^2 = \|A\|_2^2.\end{aligned} \tag{A.12}$$

Let us look at the spectral norms of two special types of matrices: orthogonal/unitary ones and self-adjoint ones. The spectral norm of an orthogonal/unitary matrix equals one. For in this case A^*A is the identity matrix and all eigenvalues of the identity are one. If $A \in \mathbb{K}^{n\times n}$ is self-adjoint, then $\|A\|_2 = \rho(A)$. Due to the spectral theorem every self-adjoint matrix is diagonalizable by orthogonal/unitary matrices. Moreover, the eigenvalues of A are real. Hence

$$\|A\|_2^2 = \rho(A^*A) = \rho(A^2) = \rho(U\Lambda^2U^*) = \max_{1\le i\le n} \lambda_i^2 = \rho(A)^2.$$

The most important matrix norm which is not an induced norm is the *Frobenius norm.* It corresponds to the Euclidean norm for vectors and is defined by

$$\|A\|_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2\right)^{\frac{1}{2}}.$$

A.9 Systems of linear equations

A general system of m linear equations in n unknowns has the form

$$\begin{array}{ccccc} a_{11}x_1 + & \cdots & + a_{1n}x_n & = & b_1 \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1}x_1 + & \cdots & + a_{mn}x_n & = & b_m \end{array}$$

Using matrix-vector notation we can write this simply as $Ax = b$ with system matrix $A \in \mathbb{K}^{m\times n}$, vector of unknowns $x \in \mathbb{K}^n$ and right-hand side $b \in \mathbb{K}^m$. A system with more equations than unknowns ($m > n$) is called *overdetermined.* If there are less equations than unknowns ($m < n$), then the system is *underdetermined.* The $m \times (n+1)$ matrix that results from appending b to the right side of A is called *augmented matrix.* The Rouché-Capelli theorem states that the linear system $Ax = b$ has a solution, if and only if the augmented matrix has the same rank as A. If $\operatorname{rank}(A) = n$, then the solution is unique. Otherwise there are infinitely many solutions.

When solving a system of linear equations one often interchanges columns or rows or adds a multiple of a row to another. Such operations can be written as matrix-matrix multiplications, which proves particularly useful when analysing algorithms for solving systems of linear equations. While row operations correspond to multiplication of the (augmented) matrix with another matrix from the

left, column operations correspond to multiplication of A with another matrix from the right.

For example, interchanging columns k and ℓ can be realized by multiplying from the right with an identity matrix that has columns k and ℓ interchanged. Such a matrix is also called *permutation matrix*. Multiplying the same matrix from the left interchanges rows k and ℓ. Multiplying a row by $\alpha \in \mathbb{K}$ is realized by multiplication from the left with an identity matrix that has the corresponding 1 replaced wit α. Finally, adding α times row k to row ℓ can be achieved by multiplying from the left with an identity that has an α inserted at position (ℓ, k).

In general, finding a matrix that performs a certain succession of row operations on A can be done by appending an identity matrix horizontally to A (i. e. to the left or right of A) and performing the row operations on the augmented matrix $(I \mid A)$ or $(A \mid I)$. The modified identity matrix is just the matrix that, if multiplied from the left with A, performs the desired row operation. For column operations the identity matrix must be appended vertically to A.

Example. Suppose we want to perform a series of row operations on the matrix

$$A = \begin{bmatrix} & & 6 \\ & 1 & 2 \\ 1 & 5 & 3 \end{bmatrix}.$$

Interchanging rows one and three, for instance, can be achieved by multiplying A from the left with a 3×3 identity matrix that has columns 1 and 3 interchanged:

$$\begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix} \begin{bmatrix} & & 6 \\ & 1 & 2 \\ 1 & 5 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 3 \\ & 1 & 2 \\ & & 6 \end{bmatrix}.$$

Next we want to divide the third row by six. This can be realized by multiplication from the left with the matrix $\operatorname{diag}(1, 1, 1/6)$:

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{1}{6} \end{bmatrix} \begin{bmatrix} 1 & 5 & 3 \\ & 1 & 2 \\ & & 6 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 3 \\ & 1 & 2 \\ & & 1 \end{bmatrix}.$$

Finally, we eliminate the 3 in the top right of the resulting matrix by subtracting three times the third row from the first. Using matrix multiplications this reads as

$$\begin{bmatrix} 1 & & -3 \\ & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 5 & 3 \\ & 1 & 2 \\ & & 1 \end{bmatrix} = \begin{bmatrix} 1 & 5 & \\ & 1 & 2 \\ & & 1 \end{bmatrix}.$$

The matrix that does all three row operations at once, that is, the matrix R satisfying

$$RA = \begin{bmatrix} 1 & 5 & \\ & 1 & 2 \\ & & 1 \end{bmatrix}$$

is given by the product of the three matrices

$$R = \begin{bmatrix} 1 & & -3 \\ & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & & \frac{1}{6} \end{bmatrix} \begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & & 1 \\ & 1 & \\ & \frac{1}{6} & \end{bmatrix}.$$

The same matrix is recovered by performing the three row operations on A with an identity appended horizontally.

$$\left[\begin{array}{ccc|ccc} 1 & & & & & 6 \\ & 1 & & & 1 & 2 \\ & & 1 & 1 & 5 & 3 \end{array}\right] \to \left[\begin{array}{ccc|ccc} & & 1 & 1 & 5 & 3 \\ & 1 & & & 1 & 2 \\ 1 & & & & & 6 \end{array}\right] \to$$

$$\left[\begin{array}{ccc|ccc} & & 1 & 1 & 5 & 3 \\ & 1 & & & 1 & 2 \\ \frac{1}{6} & & & & & 1 \end{array}\right] \to \left[\begin{array}{ccc|ccc} -\frac{1}{2} & & 1 & 1 & 5 & \\ & 1 & & & 1 & 2 \\ \frac{1}{6} & & & & & 1 \end{array}\right]$$

Appendix B

The Arnoldi and Lanczos Procedure

We consider an arbitrary matrix $A \in \mathbb{R}^{n\times n}$ and a given residual vector $r_0 \in \mathbb{R}^n$ with the corresponding Krylov subspace ($m \leq n$)

$$\mathcal{K}_m = \mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\}. \tag{B.1}$$

Let the matrix

$$K_m = [r_0|Ar_0|\cdots|A^{m-1}r_0] \in \mathbb{R}^{n\times m} \tag{B.2}$$

be the corresponding *Krylov matrix*.
The Arnoldi procedure is based on the Gram-Schmidt algorithm to construct an orthonormal set/basis of the column vectors of K_m. The basic idea is to start with $v_1 = r_0/\|r_0\|_2$, then in the j-th step multiply the current Arnoldi vector v_j with A and orthonormalize Av_j against all previous Arnoldi vectors. If the plain Gram-Schmidt procedure is used for orthonormalization, this leads to the following steps

$$\begin{aligned}
& & v_1 &= r_0/\|r_0\|_2 \\
w_1 &= Av_1 - (v_1^*Av_1)v_1, & v_2 &= w_1/\|w_1\|_2 \\
w_2 &= Av_2 - (v_1^*Av_2)v_1 - (v_2^*Av_2)v_2, & v_3 &= w_2/\|w_2\|_2 \\
& & &\vdots \\
w_m &= Av_m - (v_1^*Av_m)v_1 - \ldots - (v_m^*Av_m)v_m, & v_{m+1} &= w_m/\|w_m\|_2
\end{aligned} \tag{B.3}$$

This generates the Arnoldi matrix

$$V_m = [v_1|v_2|\cdots|v_m] \in \mathbb{R}^{n\times m} \tag{B.4}$$

and the upper Hessenberg matrix $\hat{H}_m \in \mathbb{R}^{(m+1)\times m}$

$$\hat{H}_m = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1m} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2m} \\ & h_{32} & h_{33} & & h_{3m} \\ & & \ddots & \ddots & \vdots \\ & & & h_{m(m-1)} & h_{mm} \\ & & & & h_{(m+1)m} \end{bmatrix}$$

$$= \begin{bmatrix} v_1^*Av_1 & v_1^*Av_2 & v_1^*Av_3 & \dots & v_1^*Av_m \\ v_2^*Av_1 & v_2^*Av_2 & v_2^*Av_3 & \dots & v_2^*Av_m \\ & v_3^*Av_2 & v_3^*Av_3 & & v_3^*Av_m \\ & & \ddots & \ddots & \vdots \\ & & & v_m^*Av_{m-1} & v_m^*Av_m \\ & & & & v_{m+1}^*Av_m \end{bmatrix}.$$

Note that the off-diagonal elements of $\hat{H}_m$ are calculated in the normalization step of (B.3) by calculating $\|w_m\|_2 = v_{m+1}^* w_m = v_{m+1}^* Av_m$ using the mutual orthogonality of the Arnoldi vectors, i.e., $v_{m+1}^* v_j$, $j = 1, \dots, m$.
If a breakdown occurs in the m-th step, $w_m = 0$ is defined but not v_{m+1} and so the algorithm stops; in this case the element $h_{m+1,m} = 0$.

Lemma B.1. *If the Arnoldi iteration (B.3) does not terminate prematurely, the vectors $v_1, \dots, v_m$ form an orthonormal basis of the Krylov space $\mathcal{K}_m$, i.e., $V_m^* V_m = I_m$. Further, the operator $P_m = V_m V_m^*$ is the orthogonal projection onto $\mathcal{K}_m$.* □

In any case, we have from (B.3) that

$$Av_j = w_j + \sum_{i=1}^{j} h_{ij} v_i, \tag{B.5}$$

which means in matrix notation [1]

$$AV_m = V_m H_m + w_m e_m^* \ (= V_{m+1}\hat{H}_m), \tag{B.6}$$

with $H_m \in \mathbb{R}^{m\times m}$ is the square Hessenberg matrix obtained from $\hat{H}_m$ by removing the last row and e_m denotes the m-th unit vector.

Lemma B.2 (Arnoldi identity)**.** *From (B.6) we conclude that*

$$V_m^* AV_m = H_m + \underbrace{V_m^* w_m}_{=0} e_m^* = H_m. \tag{B.7}$$

[1] Note for the last identity that $V_{m+1}\hat{H}_m = [V_m | v_{m+1}] \begin{bmatrix} H_m \\ (v_{m+1}^* Av_m) e_m^* \end{bmatrix} = V_m H_m + (v_{m+1}^* Av_m) v_{m+1} e_m^*$ and due to orthogonality $v_{m+1}^* Av_m = \|w_m\|_2$ (use (B.3)).

Proof. We have $V_m^* A V_m = H_m + \underbrace{V_m^* w_m}_{=0} e_m^* = H_m$. □

Theorem B.1 ([2]). *The Arnoldi procedure generates a reduced QR-decomposition of the Krylov matrix K_m in the form*

$$K_m = V_m R_m \tag{B.8}$$

with the Arnoldi matrix V_m from (B.4) and a triangular matrix $R_m \in \mathbb{R}^{m \times m}$. Further, the Hessenberg matrix H_m is an projection of A onto the Krylov subspace spanned by the columns of K_m, i.e.,

$$H_m = V_m^* A V_m. \tag{B.9}$$

The Arnoldi procedure is usually implemented in a numerically stable way known as *Modified Gram-Schmidt algorithm* (MGS).

The Arnoldi iteration in the modified Gram-Schmidt variant is summarized in Alg. 10.

Algorithm 10: Arnoldi iteration (Modified Gram-Schmidt variant)

```
Data: Matrix A, vector r_0, number m
Result: Arnoldi vectors v_1, ..., v_m, Upper Hessenberg matrix
        Ĥ_m = (h_ij)
Initialization: v_1 ← r_0/‖r_0‖_2
for j = 1...m do
    w_j ← A v_j
    for i = 1...j do
        h_ij ← v_i^* w_j
        w_j ← w_j − h_ij v_i
    end
    h_(j+1),j ← ‖w_j‖_2
    if h_j+1,j = 0 then
        Stop
    end
    v_j+1 ← w_j/h_(j+1),j
end
```

In the case where A is symmetric, also the upper Hessenberg matrix H_m is symmetric ($H_m = V_m^* A V_m = V_m^* A^* V_m = H_m^*$) and therefore a symmetric tridiagonal matrix

$$T_m = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \beta_m \\ & & & \beta_m & \alpha_m \end{bmatrix}.$$

As a consequence, the inner loop in Alg. 10 simplifies, see Alg. 11. This is called the *Lanczos iteration.*

Algorithm 11: Lanczos iteration

Data: Matrix A symmetric, vector r_0, number m
Result: Arnoldi vectors $v_1, \ldots, v_m$, tridiagonal matrix T_m
Initialization: $\beta_1 \leftarrow 0$, $v_0 \leftarrow 0, v_1 \leftarrow r_0/\|r_0\|_2$
for $j = 1 \ldots m$ **do**
  $w_j \leftarrow Av_j - \beta_j v_{j-1}$
  $\alpha_j \leftarrow v_j^* w_j$
  $w_j \leftarrow w_j - \alpha_j v_j$
  $\beta_{j+1} \leftarrow \|w_j\|_2$
  if $\beta_{j+1} = 0$ **then**
    Stop
  end
  $v_{j+1} \leftarrow w_j/\beta_{j+1}$
end

Bibliography

[1] M. Abramowitz and I.A. Stegun, eds. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* US Government Printing Office, 1972.

[2] W. Auzinger and J.M. Melenk. *Iterative Solution of Large Linear Systems.*. Lecture notes, TU Wien, 2009.

[3] J.-P. Berrut and L.N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review, Vol. 46, No. 3, pp. 501–517.* SIAM, 2004.

[4] B. Cipra. The Best of the 20th Century: Editors Name Top 10 Algorithms. *SIAM News, Vol. 33, No. 4.* SIAM, 2000. https://archive.siam.org/pdf/news/637.pdf

[5] L. Exl. *Numerical Methods II (Computational Science)* Lecture notes, Uni Wien, 2021.

[6] G.H. Golub and C.F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 2013

[7] M. Hanke-Bourgeois. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens.* Vieweg+Teubner, 2009.

[8] N.J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 2002.

[9] C.F. Van Loan. *Introduction to Scientific Computing.* Prentice Hall, 1997.

[10] J. Nocedal and S. Wright. *Numerical Optimization - Second Edition.* Springer Verlag, Science & Business Media, 2006.

[11] F.W.J. Olver et al., eds. *NIST Digital Library of Mathematical Functions.* http://dlmf.nist.gov/

[12] R. Plato. *Concise Numerical Mathematics.* American Mathematical Society, 2003.

[13] Y. Saad. *Numerical Methods for Large Eigenvalue Problems - Second Edition.* SIAM 2011.

[14] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis.* Springer, 2002.

[15] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2016.

[16] L.N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[17] www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html

[18] http://fourier.eng.hmc.edu/e161/lectures/algebra/node7.html